

A software engineering guideline for self-organizing 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. 2010. "A software engineering guideline for self-organizing resource-flow systems." In *2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems, 27 September - 1 October 2010, Budapest, Hungary*, edited by Márk Jelasity, Janos Sztipanovits, Indranil Gupta, Salima Hassas, and Jerome Rolia, 194–203. Piscataway, NJ: IEEE.
<https://doi.org/10.1109/saso.2010.26>.



A Software Engineering Guideline for Self-organizing Resource-Flow Systems

Hella Seebach, Florian Nafz, Jan-Philipp Steghöfer, Wolfgang Reif
Institute for Software & Systems Engineering
Universität Augsburg, Universitätsstr. 6a, D-86159 Augsburg
{seebach, nafz, steghoefer, reif}@informatik.uni-augsburg.de

Abstract—When introducing self-organization into a system, its developer aims to reduce the system’s complexity, during development as well as during operation. More often than not, the self-organization mechanism is ingenious, highly tweaked for the system under construction and not reproducible or reusable by other developers or in other projects.

This paper introduces a software engineering guideline for self-organizing resource-flow systems along with an elaborated pattern that describes the elements of the system under construction and their collaboration. Together, guideline and pattern are the basis for a well-defined approach for the design and construction of systems in this class, which includes, among others, logistics applications, and adaptive production systems. They therefore allow developers to achieve reproducible results within a documented design framework, leverage the possibilities of the underlying formal approach and reuse self-organization mechanisms tailored for the system class. The paper demonstrates the application of the guideline with a running example.

Keywords-software engineering; design methodology; pattern; self-organization;

I. INTRODUCTION

Traditional engineering usually fails when the system to be engineered is not “traditional”. This becomes especially obvious with self-organizing systems which are usually the result of a creative development process and often apply bio-inspired mechanisms that are tailored to a specific system’s needs. They allow a system to achieve self-organization of the system’s components and thus desirable qualities like self-adaptivity, self-healing, etc. However, such specific solutions are mostly not applicable to even slightly different problems, as the creative process is rarely embedded in a defined engineering methodology suitable for this kind of systems. Many argue that we are in a transitional phase at the moment where the discipline of software engineering shifts towards new paradigms and techniques, most of which are still unclear [1].

During this transition however, the power of self-organization has to be made available by means that a software engineer can employ right now, with knowledge and expertise that is available to a broad public at the moment. This is where projects like SodekoVS [2] and SelfLet [3] come in that try to make the new world accessible with the old tools. The software engineering guideline based on the *Organic Design Pattern* or ODP – first introduced in [4] and revised and elaborated here – is another such mean: it employs

traditional software engineering and formal techniques to describe a system that has some or all of the self-x properties mentioned above.

It has been recognized early on that there is no such thing as a “one-size-fits-all” software engineering process. Therefore, method fragments have been proposed [5] that can be used to adapt an existing process with elements from other processes for the specific requirements of a project. Later work extended this idea to the domain of agent-oriented software engineering [6] and introduced multi-agent specific ideas. The steps and artifacts proposed in this paper could be called method fragments as well, however apart from their description and their definition with the Software Process Engineering Meta-Model (SPEM)¹, we also show how they are related and how they interact. Therefore, we are not only proposing fragments but a guideline that enables software engineers to enhance classical software engineering processes like the Unified Process with the necessary steps for the design and construction of self-organizing resource-flow systems. The tools (e.g. UML, OCL, Model Checker) used in the activities of the guideline which enable the dynamic reconfiguration are well known.

Thus, the presented techniques close the gap between the two worlds of classical software engineering and self-organizing systems.

II. OVERVIEW

In the Organic Computing (OC) community, research on systems that behave in a life-like fashion is conducted. Such systems are often called *Organic Computing systems* or *self-x systems*. In many cases, self-organization is a fundamental property of such systems that yields the life-like behaviour. The SAVE ORCA project regards self-x systems from the vantage point of the software engineer and provides a formal background for analysis and verification of such systems. This makes it possible to use self-organization in applications that are developed in non-academic settings and that require formal analysis, e.g., in case certification of the product is required.

The guideline presented here is designed to tailor an existing SE process to be used for self-organizing resource-flow systems. It allows to systematically enhance traditional

¹<http://www.omg.org/technology/documents/formal/spem.htm>

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.

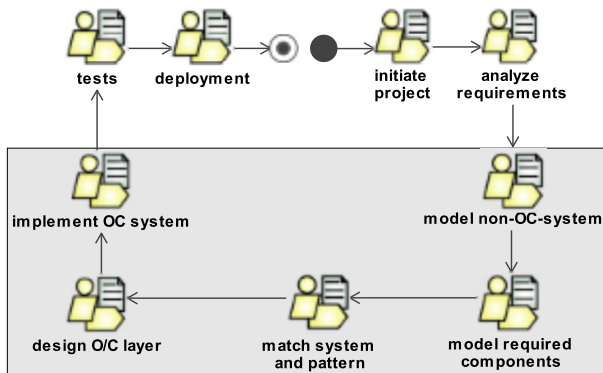


Figure 1. Guideline in SPEM notation

In this section, the guideline and its activities are sketched before they are fully elaborated in Section V. Figure 1 depicts a SPEM model of the guideline. The highlighted activities are novel and will be elaborated in the following while the others are well known classical software engineering activities. The novel steps are supported by artifacts and techniques developed in the SAVE ORCA project. After initiating the whole project and analyzing the requirements for the self-organizing resource-flow system, the guideline instructs the engineer to *model the non-OC-system*. This means, if a productive system already exists, this system must be investigated and the corresponding concepts must be modeled. If not, the requirements must be 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-organization 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. Then the activity *match system and pattern* follows which encapsulates the mapping of traditional system components to components defined by the pattern presented in this paper. After this, the activity *design O/C layer* follows which determines the way the system self-organizes and enables the engineer to make statements about the self-organizing behavior of the new system. Finally, the

self-organizing resource-flow system is implemented (*implement OC system*) in a runtime environment. The guideline ends with traditional tests and deployment activities.

For a better understanding of the individual steps of the SE guideline, the next two sections introduce the class of resource-flow systems and one case study (Section III) which illustrates the concepts of the SE guideline throughout the paper and the Organic Design Pattern (Section IV) which is an important aspect of the guideline. The ODP is described in a standardized pattern format and will be used extensively in the *match system and pattern* activity. Then, Section V presents the software engineering guideline with every activity and step in detail. Additionally, the SE guideline is applied to the case study. Section VI points to other research in this area. Finally, Section VII concludes the paper and sketches future work.

III. RESOURCE-FLOW SYSTEMS EXEMPLIFIED IN AN ADAPTIVE PRODUCTION CELL

Resource-flow systems in general are systems producing, processing and consuming resources in a given order. They consist of several stations (machines, agents) which are able to change the state of the resource. The way the resources take along the several stations is called resource-flow. In the class of self-organizing resource-flow systems we consider systems where the processing stations process the resources in a given sequential order and the resources are not split up or merged. One prominent example for such resource-flow systems is the production of cars, especially the assembly of car bodies.

In the following a case study in this domain is described, which illustrates how the software engineering guideline presented in this paper can be employed in the domain of production automation systems. The example is an adaptive production cell the way it could be designed in the future. Traditional engineering designs production cells in a very static manner. Individual machines that process a workpiece are linked with each other in a strictly sequential order by conveyors or similar mechanisms. The layout of the cell is therefore predefined, very inflexible, and rigid. Additionally – and maybe most importantly – such a system is extremely error-prone as failure of one component will bring the whole line to a standstill.

The vision of an adaptive production cell presented here consists of machines (in this case robots) which are capable of using different tools and which are connected with flexible, autonomous transportation units (carts). The functional goal of the cell is to process workpieces according to a given specification, the *task*. A task is a user-defined sequence of tool applications.

The example system considers a scenario for the assembly of cars. It contains five robots (A-E) and six autonomous carts (1-6). Each robot has several out of five distinct capabilities: weld the body (B), attach the cables (Ca), insert

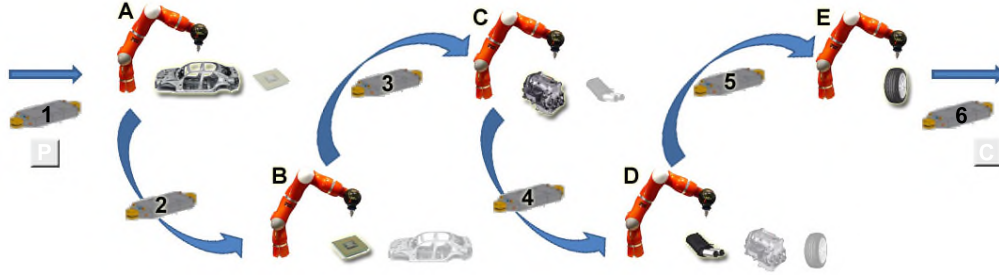


Figure 2. Car assembly scenario

the motor (M), attach the exhaust (E,) and finally the wheels (W). The capabilities of robots are defined by properties of the individual robots (degrees of freedom, payload, etc.) and their access to the material supply (e.g. cables, wheels). Carts transport workpieces between the robots. Further, all carts have a *produce*-capability (P) and a *consume*-capability (C), meaning that they are able to introduce resources to the production cell or remove completely assembled cars again.

In the scenario depicted in Figure 2, every workpiece must be processed by all five tools in a given order (plus produce and consume): 1st: body, 2nd: cables, 3rd: motor, 4th: exhaust, 5th: wheels (short: PBCaMEWC). For the initial configuration of the system, the capabilities required for the task are distributed among the robots, such that no robot has to switch tools. Further, the transportation of workpieces is organized accordingly. In the figures, the capabilities of the carts are omitted for the sake of better readability. The capabilities that are currently applied are shown as small icons next to the robots where the unused but available capabilities are grayed out. Cart1 and Cart6 apply produce and consume respectively. Which capability is applied, where the resource comes from and who to give the resource to is determined by the *role* of a cart or robot.

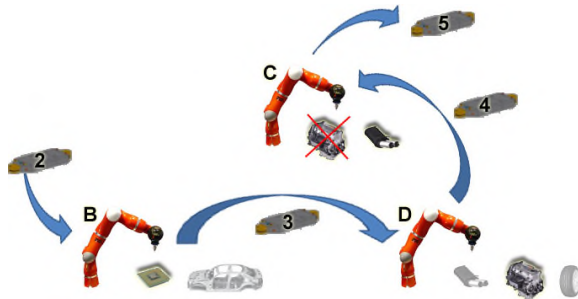


Figure 3. Section of reconfigured system

The distributed redundancy allows for reconfiguration even if a robot or cart becomes entirely unavailable. If a capability – e.g. the capability of RobotC to insert the motor – is not available anymore, the system reconfigures to find a new distribution of capabilities and new routes for the carts such that assembly can continue. In Figure 3, the system

has been reconfigured in comparison to Figure 2 because RobotC has lost its capability to insert the motor. RobotD is now configured to insert the motor and RobotC adopted the assignment from RobotD and attaches the exhaust. The carts have also been reconfigured. Cart3 now has to deliver the workpiece to RobotD instead of RobotC and Cart4 and Cart5 are adapted accordingly. Of course, there are more complex reconfiguration scenarios conceivable which are not shown here, like graceful degradation (e.g. one robot has to apply more than one capability or a cart has to transport car bodies between several robots). In each situation the system has to reconfigure in a correct manner, so that processing can continue as long as every required capability is still available at least once in the system.

IV. ORGANIC DESIGN PATTERN

In this section the Organic Design Pattern (ODP) is presented which is the key component of the guideline presented in Section V. The notation of the ODP follows the pattern notation of Gamma et al. [7]. For better readability the order of the sections has been altered.

Intent: The ODP is a design principle for a broad but defined class of self-x systems, namely those which consist of a set of independent components interacting with each other to process resources and where reconfiguration and adaptation respectively can be expressed as a reallocation of roles. It provides the missing link between the world of self-organizing systems and classical software engineering methods.

Motivation: The complexity of software and the requirements for software systems increase steadily. The Organic Computing community aims to build systems which have the capability to autonomously (re-)organize and adapt themselves. This introduces self-x properties (e.g. self-healing) which have several benefits: systems become more dependable, as they can compensate for some failures; systems become easier to maintain, because they can automatically configure themselves; systems become more convenient to use because of automatic adaptation to new situations. The ODP is a tool to simplify the challenging task of designing and constructing systems with self-x properties. The core of the pattern, an elaborate role concept, allows

to model systems that adapt to changing tasks and process several resources with different tasks at the same time. Furthermore, systems designed with ODP maintain their functionality as long as possible, even if that means that they have to run in a *degraded mode* where a task is only partially fulfilled or an agent has to apply several roles for the same task. The underlying formal model allows the formulation of behavioral guarantees and their monitoring at runtime [8].

Applicability: Use the Organic Design Pattern when

- you have a system with a resource flow,
- you have a system with several components that cooperate to fulfill a task. These components have a degree of redundancy in their capabilities/inputs/outputs or the components themselves are redundant in the system,
- you want to upgrade a traditional system to a self-organizing system for more reliability and easier maintenance,
- you want to give behavioral guarantees despite the self-organizing behavior of the system.

Participants: The following classes and objects participate in the ODP:

- **Agent** is one component of the system.
- **Capability** are the abilities an agent can apply to a resource. The *produce*-capability denotes the starting point of the resource flow, *process* processes the resource during the resource flow, and *consume* denotes the endpoint of the resource flow.
- **Resource** can, e.g., be a workpiece or a data object that has to be processed according to its task by the system.
- **Task** is the sequence of capabilities that have to be applied to the given resource. Always starts with a produce capability and ends with a consume capability.
- **Role** determines which capabilities an agent has to apply to the given resource. A role has pre- and postconditions.
- **Condition** either determines from which agents the resource originates and which state and task the resource has (precondition) or to which agent the resource has to be given to and which task and state the resource has (postcondition).
- **Observer/Controller** is a container for the specification of the reconfiguration algorithm (which calculates the role allocations), as well as the monitoring and controlling strategies.

Structure: The construction model seen in Figure 4 is instantiated for each domain and instances of the system in the domain can be defined. This is part of the guideline as outlined in Section V. The static parts of an ODP system are shown in Figure 4 as an UML class diagram. The main concept of an ODP system is the *agent*. According to a given *task* the *agent* processes *resources* with one or more of its *capabilities*. A *task* describes how a given *resource* should be processed. It is an ordered and non-unique sequence

of *capabilities* which have to be applied to the *resource*. The *capabilities* can be distinguished between three types: *capabilities* that *produce*, *consume* and *process resources*. Thus, *agents* with a *produce-capability* are sources which introduce resources into the system and are thus the starting point of the resource flow. *Agents* with a *consume* capability are analogously the sinks and endpoints of the resource flow. Every *agent* is characterized by the *capabilities* it can provide and the *agents* to which/from which it can give/receive *resources*. These possible interactions on a *resource* level are given by the *inputs/outputs*-relations of the *agents*. These I/O-relations form the so called *I/O graph*.

An *agent* can have several *roles*. The mapping of *roles* to *agents* is called *role allocation* and is represented in *allocatedRoles*. Self-organization in the system class is described as a role allocation problem. The selected *role* of an *agent* determines which *capabilities* the *agent* performs in a specific situation. A *role* consists of a *precondition*, a sequence of *capabilities* that need to be applied and a *postcondition*. The *precondition* describes which *resources* are accepted by the *agent* and which other *agent* provides them (*port*). The *postcondition* describes the *resource* state after processing and which *agent* should receive it. *Conditions* are 3-tuples of a target *agent* 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. For example, the role allocated to RobotA in the case study looks like this:

```
Precondition: (Cart1, P, PBCaMEWC)
Capabilities to Apply: B
Postcondition: (Cart2, PB, PBCaMEWC)
```

The resource is taken from Cart1 with status P and task PBCaMEWC, the capability “weld body” (B) is applied and the resource is then given to Cart2 with the new status PB and the task PBCaMEWC.

The *resource-flow 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. If all resource-flow graph for all tasks in a system are combined, the *role allocation graph* of the whole system is formed.

Several OCL constraints [9] are used to ensure a number of system properties. They are, e.g., used to determine the structure of the role allocation graph. In [10] some constraints are listed, divided into monitoring and consistency constraints. Monitoring constraints have to be monitored at runtime by the agents themselves or by a central observer instance. Consistency constraints define a correct role allocation and thus form a part of the specification of the reconfiguration algorithm. In Figure 4, the constraints can be seen (in a short notation) in the classes *agent*, *task* and *role*. An example for a consistency constraint is the **I/O-Consistency** for an agent (self):

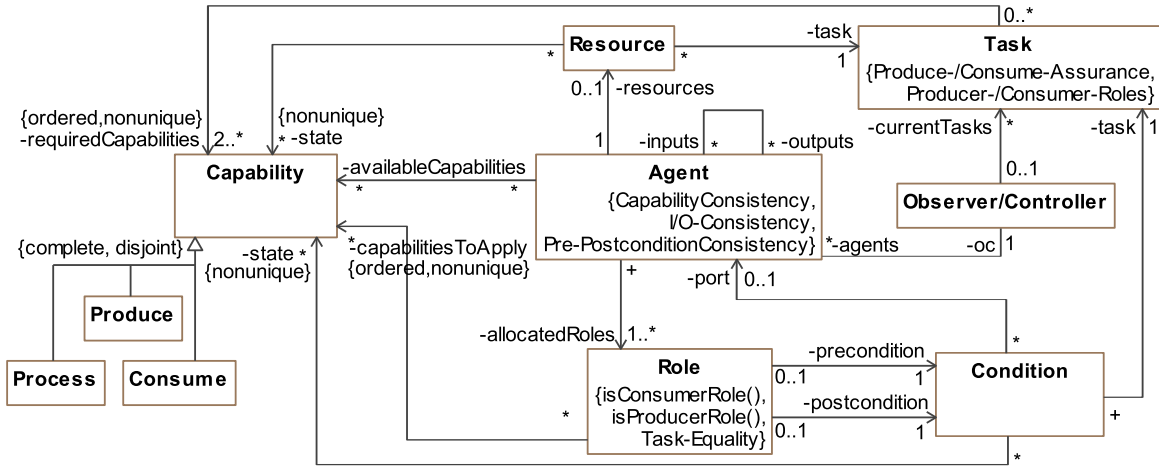


Figure 4. Organic Design Pattern - Construction Model

```
(self.inputs -> includesAll(
  self.allocatedRoles.precondition.port))
and (self.outputs -> includesAll(
  self.allocatedRoles.postcondition.port))
```

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 can be monitored and is violated in case an agent is no longer responsive.

The reconfiguration mechanism is encapsulated in the concept of the *observer/controller* (O/C). The output of the reconfiguration algorithm is a new allocation of *roles* to *agents* that restores the system to a state in which it conforms to the constraints and is able to fulfill its tasks. The constraints and static components of the ODP govern the output of the reconfiguration algorithm. The classes, relations and cardinalities along with the OCL constraints specify the valid results and which role allocations are correct and valid for use in a system modeled with ODP.

Collaborations: One goal of this pattern is to design systems with self-x properties and nevertheless give behavioral guarantees. For that purpose, the behavior of the system components, here the agents, must also be investigated and finally verified. Hence, the pattern contains a clear definition of the agents' interaction behavior and their behavior with regard to resource handling in form of state machines and communication protocols. These behaviors however are only described briefly here and will be detailed in a forthcoming paper.

The behavior of an agent is subdivided into the productive or functional behavior and the reconfiguration behavior. This separation is done by the use of two subagents. The agent for the functional behavior is defined once for a system class while the reconfiguration agent acts as an interface where different reconfiguration mechanisms can be plugged in. Each agent has limited knowledge about its immediate sur-

roundings. The I/O-relation and the role allocation allow to construct an excerpt of the I/O graph and the role allocation graph which contains the direct neighbors of the agent. This knowledge can, e.g., be used in a distributed reconfiguration mechanism. Additionally, the separation allows to integrate different concepts, like, e.g., the one presented in [11] with ODP and gives the opportunity to devise and reuse different (central or decentral) reconfiguration mechanisms. The choice of the most suitable reconfiguration mechanism for each domain is one part of the guideline as described in Section V.

Consequences: The Organic Design Pattern has the following benefits:

- makes it easy to add new agents and tasks to a system
- allows the intuitive abstract description of the reconfiguration process (O/C)
- grants precisely defined self-x properties once the system has been modeled with the help of the ODP
- enables safety analysis and formal verification of the system's properties and their measurement on a generic level.

The most important idea behind these consequences is the so called *Restore Invariant Approach* which is explained in detail in [8]. This technique allows to formulate a behavioral corridor constraining the system behavior to the desired behavior. The system can evolve arbitrarily but has to fulfill invariants (formalized as OCL constraints) and therefore stays within the corridor. Whenever a failure occurs and the constraints are violated, the system needs to reconfigure and restore the constraints. If the engineer only describes the domain-dependent behavior, e.g. the concrete execution of capabilities, the added aspects will not interfere with the system's behavior in a manner which violates the verified model. To enable this technique the system has to be specified and modeled with the presented pattern.

Implementation and Sample: The ODP is provided in the context of a software engineering guideline which defines the concrete use of the pattern. In Section V the recommended software engineering guideline including the ODP is applied to the case study of an adaptive production cell. A reference implementation for systems modeled with this pattern according to the guideline presented in this paper is given in [12].

V. SOFTWARE ENGINEERING GUIDELINE

This section describes a software engineering guideline that incorporates the ODP and enables software engineers to develop self-organizing resource-flow systems. In the following, the SE guideline depicted in Figure 1 is explained step by step in detail and applied to the running example. Every activity of the guideline is modeled with its artifacts and included tasks in SPEM. Due to space limitations only the activities *Match System and Pattern* and *Design O/C-Layer* are shown in complete detail (see Figures 5 and 8). The other activities are described only textually. The SPEM models and an integration of the guideline with the Unified Process are available at <http://saveorca.isse.de>.

As shown in Figure 1, the OC-specific activities start at a point in the software engineering process where a traditional requirement analysis has already taken place. The first activity consists of one task:

1. Model the concepts of a traditional non-OC-system (*model non-OC-system*). This means that either a traditional resource-flow system already exists and should be enhanced by self-x behavior or a new system has to be built. In the former case, the existing system must be investigated and the concepts modeled, while in the latter the requirements analysis and analysis phase of a classical software engineering process will result in a conceptual model.

Based on this conceptual model the guideline continues with the following three tasks as part of the activity *model required components*:

2. Identify the components and their collaboration structure of the system under construction. In the case study, among others, robots and workpieces are components.
3. Identify groups of similar components. These groups will later become instances of the class *agent* of the Organic Design Pattern. This will be robots and carts in our example.
4. Add degrees of freedom to the system. This means either adding additional (in most cases redundant) capabilities to the components so that they are able to adapt to changing requirements or enhancing the possibilities of the resource flow (I/O-relations), so that there exist several ways for one resource to be delivered. In the example, the degree of freedom in the system results from the possibility to connect several

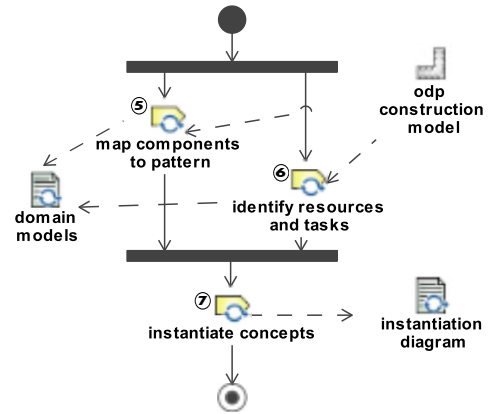


Figure 5. Match System and Pattern

robots by carts and that the robots have different and redundant capabilities.

The next three tasks of the guideline cover the activity *match system and pattern*. As seen in Figure 5 the artifact *odp construction model* is a guidance and input artifact for the tasks *map components to pattern* and *identify resources and tasks*. The *domain models* and an *instantiation diagram* are output artifacts. Examples of these output artifacts are shown in Figure 6 and Figure 7.

5. Map the concepts of the traditional system or analysis to the concepts of the ODP. This mapping process makes the components “OC-ready”. The obtained self-x infrastructure has three main benefits. First, it wraps the components into agents which can announce their capabilities and are aware of their state. This means they are able to reflect on themselves and can announce wrong behavior or broken capabilities etc. Second, the agents have a well formed communication infrastructure. Third, the agents gain the ability to use their capabilities according to the actually chosen role and to update their allocated roles if requested by an observer/controller. Figure 6 (the domain model for robots) shows this mapping for the case study and the domain of production automation. It also shows the mapping of the concepts *resource* and *task* (done in task six of the guideline). A *robot* is an ODP *agent* and can have the capabilities *motor*, *cables*, *wheels*, *body* and *exhaust* which are all *process* capabilities. It processes a *workpiece* according to its *task*. Like the agent in the ODP, the robot has allocated *roles* with according *conditions*. For the case study, there is also a domain model for *carts* which are also ODP *agents* and can have the capabilities *produce* and *consume*.
6. Identify the resources and tasks in the traditional or analyzed system. As seen in Figure 6, the resources in the production cell are the workpieces and the tasks are composed of the required capabilities.

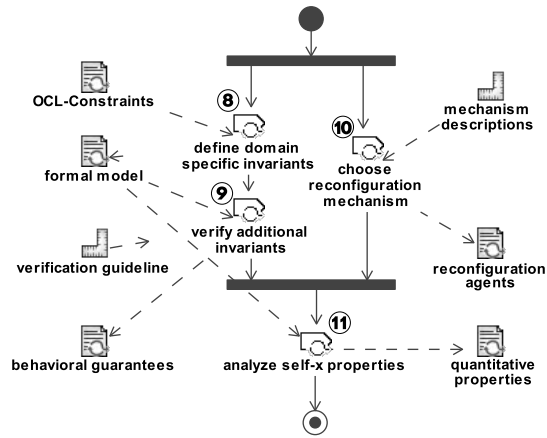


Figure 8. Designing the OC-Layer

is then used as input for the constraint solver Kodkod [13] which calculates a new correct role allocation.

If decentralized reconfiguration is required, there exists a mechanism for a wave-like reconfiguration [14]. The idea here is that an agent recognizing an invariant violation asks its neighbor corresponding to the executed role for help. If the neighbor is able to help, the two agents exchange their roles, otherwise the neighbor asks its neighbor and so on until some agent can help the agent where the invariant violation was originally detected. Another promising approach exists with coalition formation based on the resource flow and role allocation graphs which will be published elsewhere. The mechanisms are easily integrable into the implementation (see tasks 12–14 of the SE guideline).

11. (Optional) Analyze self-x properties. In [15] a technique is described which allows to quantify the improvements of an self-x system with regard to reliability and failure tolerance in comparison to a traditional one. This task requires the *formal model* (input artifact) and yields the *quantitative properties* (output artifact).

After designing all elements of the self-organizing resource-flow system the activity *implement OC system* follows. Another benefit of the presented approach is worth emphasizing. Most of the dynamic agent behavior and other crucial parts of the system are already generically implemented in the ODP Runtime Environment (ORE, see [12]). It is generic in a sense that it can be used for all systems modeled with the ODP and created with the help of the guideline, as the agent behavior and interaction protocols as well as the reconfiguration mechanism are usable for all instances. Of course, some parts still have to be adapted to the domain and application:

12. Generate Agent Definitions. The *domain models* (input artifact) and the *instantiation diagram* (input artifact) for classes derived from Agent (here: robots and carts) provide the basis for the automatic generation of *agent*

definition files (output artifact) 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.

13. Implement Capability Application. Each domain processes the resources differently. In the example’s production cell, the robots add parts to a car body, thus changing its status. At least, the change of the resource’s status has to be implemented by the engineer. In many examples, additional processing, calculations or the physical manipulation of the resource has to be implemented as well.
14. Implement Physical Interactions. 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 mostly coupled to the application of capabilities and the interaction protocols of the agents, these steps are strongly interrelated 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-organizing resource-flow systems systematically and reproducibly. Additionally, he benefits from the formal foundations that allow quantitative and qualitative statements about the developed system.

VI. RELATED WORK

Several self-organization mechanisms have been described as patterns. In [16] the authors propose patterns that mostly cover certain parts of self-organization mechanisms and are mainly suitable for situated multiagent systems (MAS). A complete approach to achieve self-organization in a multi-agent system by “infochemicals” – a generalized form of the different chemicals found in the coordination of bees – is presented as a pattern in [17]. Similarly, [18] describes gradient fields and market based control as patterns to achieve self-organization. A related paper is [19] which describes patterns for MAS that can be used for the implementation of systems that use principles of self-organization. All the papers mentioned are focused on the coordination mechanism however, while the pattern presented here explicitly describes the system’s structure and the interactions that are not related to coordination.

Many of the classical multi-agent system development processes are very much focused on the identification of artifacts and the interaction between agents. Adaptivity or self-organization is not a concern. Examples are CoMoMAS [20] and the newer Gaia [21], MaSE [22], and Tropos [23]. [24] proposes an approach that is specifically suited for the domain of flexible manufacturing systems but does not

include adaptivity as well. Some of these processes have been enhanced to include certain aspects of adaptivity [25], but none has been commonly accepted or widely adopted and with the advent of self-organizing systems the focus has shifted away from them. A survey and classification of MAS development methodologies is given in [26].

Only recently projects emerged that try to bring together the top-down software engineering of distributed systems and the bottom-up development of self-organization mechanisms. One approach to include adaptivity has been the ADELFE methodology [27]. It allows designing multi-agent systems in a dynamic environment. It applies to the first three stages of the Unified Process and covers the entire agent society as well as single agents. Changes in the MAS are considered to stem from the agents' environment rather than from the agents themselves. Guidelines for interaction of the agents or their dynamic behavior are not given.

The project SodekoVS [2] tries to unify different self-organization approaches by providing a reference architecture for a multi-agent system and a catalog of different mechanisms, along with a description of their applicability. Any fitting mechanism can then be used with the reference architecture and validated and optimized by simulation. Self-Let [3] proposes a similar approach for use in Autonomic Computing environments. However, none of the projects focus on Software Engineering or include formal analysis.

Apart from these agent-focused works, there are a number that aim at a more general engineering or design approach for complex adaptive systems. Carlos Gershenson suggests a more abstract methodology that can be applied to the description and construction of any self-organizing system (not only in computer science) [28]. At its core is the insight that such systems require exploration by simulation and refinement of preliminary models and control strategies. The description is very abstract and only hints to the properties such strategies need to have.

Jochen Fromm suggests to apply the "scientific method" to the engineering of self-organizing systems [29], i.e., creation and refinement of testable models and their evaluation according to the defined goals of the target system. Turner et al. [30] propose "rule migration", a process in which emergent properties are represented as rules of cellular automata derived from abstract concepts of higher system levels in an iterative process. Differences in the behavior of different models are then aligned to achieve equivalent representations.

All three papers mentioned above give very general advice for scientifically educated engineers, however they do not aim at software practitioners or at the integration of their methods into existing processes. This integration is tackled in [31], however the paper is very abstract and merely hints at the required steps. In comparison, the SE guideline and the pattern presented on the previous pages are very concrete and have been devised to allow the top-down construction

of self-organizing systems and to be able to make formally verified statements about them. They integrate into standard software engineering processes and can be used out of the box for highly dynamic resource-flow systems. The self-organization mechanism is described in an abstract way to enable different concrete strategies.

VII. CONCLUSION AND FUTURE WORK

This paper presented a software engineering guideline for self-organizing resource-flow systems as well as a pattern to support it and a case study to demonstrate the application of both. The approach allows the design and construction of systems that feel "organic" and allows an engineer to harness a powerful formal approach to prove properties of the system as well as different self-organization mechanisms that enable dynamic reconfiguration of the system in case of failures. The guideline and pattern can be embedded into a traditional engineering environment and are thus applicable by any trained software engineer.

In addition to the reconfiguration mechanism mentioned in this paper, we are currently working on a decentralized reconfiguration strategy that will harness the local knowledge of the agents to find minimal coalitions of agents to restore functionality in case of defects. Work on additional case studies is also in progress to investigate further aspects of the system class and the approach. Furthermore we investigate the translation of the guideline for resource-flow systems to other system classes, especial system classes where data-flows determine the behavior of the system.

ACKNOWLEDGMENT

This research is partly sponsored by the special priority program "Organic Computing" (SPP 1183) of the German research foundation (DFG) in the project SAVE ORCA.

REFERENCES

- [1] G. Serugendo, M. Gleizes, and A. Karageorgos, "Self-organisation and emergence in MAS: An overview," *Informatica*, vol. 30, no. 1, pp. 45–54, 2006.
- [2] J. Sudeikat, L. Braubach, A. Pokahr, W. Renz, and W. Lamersdorf, "Systematically Engineering Self-Organizing Systems: The SodekoVS Approach," *Electronic Communications of the EASST*, vol. 17, 2009.
- [3] D. Devescovi, E. Di Nitto, D. Dubois, and R. Mirandola, "Self-organization algorithms for autonomic systems in the SelfLet approach," in *Proceedings of Autonomics '07*. Brussels, Belgium: ICST, 2007.
- [4] H. Seebach, F. Ortmeier, and W. Reif, "Design and Construction of Organic Computing Systems," *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pp. 4215–4221, Sept. 2007.

- [5] K. Kumar and R. J. Welke, "Methodology Engineering: a proposal for situation-specific methodology construction," in *Challenges and strategies for research in systems development*. New York, NY, USA: John Wiley & Sons, Inc., 1992, pp. 257–269.
- [6] M. Cossentino, S. Gaglio, A. Garro, and V. Seidita, "Method fragments for agent design methodologies: from standardisation to research," *International Journal of Agent-Oriented Software Engineering*, vol. 1, no. 1, pp. 91–121, 2007.
- [7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison Wesley, 1995.
- [8] 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.
- [9] OMG, "Object Constraint Language, OMG Available Specification," 2006.
- [10] 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.
- [11] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer, and H. Schmeck, "Towards a generic observer/controller architecture for organic computing," *INFORMATIK 2006 – Informatik für Menschen!*, vol. P-93, pp. 112 – 119, 2006.
- [12] F. Nafz, F. Ortmeier, H. Seebach, J.-P. Steghöfer, and W. Reif, "A generic software framework for role-based Organic Computing systems," in *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2009.
- [13] E. Torlak and D. Jackson, "Kodkod: A relational model finder," *Lecture Notes in Computer Science*, vol. 4424, p. 632, 2007.
- [14] 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 *Proceedings of the 4th Workshop on Multi-Agent Systems and Simulation*, 2010.
- [15] M. Güdemann, F. Ortmeier, and W. Reif, "Formal modeling and verification of systems with self-x properties," in *Proc. of ATC-06*, 2006, pp. 38–47.
- [16] L. Gardelli, M. Viroli, and A. Omicini, "Design patterns for self-organizing multiagent systems," in *Proceedings of EEDAS 2007*, 2007.
- [17] H. Kasinger, B. Bauer, and J. Denzinger, "Design Pattern for Self-Organizing Emergent Systems Based on Digital Infochemicals," in *Proceedings of the Sixth IEEE Conference and Workshops on Engineering of Autonomic and Autonomous Systems*. IEEE Computer Society Washington, DC, USA, 2009, pp. 45–55.
- [18] T. De Wolf and T. Holvoet, "Design patterns for decentralised coordination in self-organising emergent systems," *Lecture Notes in Computer Science*, vol. 4335, p. 28, 2007.
- [19] K. Schelfhout, T. Coninx, A. Helleboogh, T. Holvoet, E. Steegmans, and D. Weyns, "Agent Implementation Patterns," in *Proceedings of Workshop on Agent-Oriented Methodologies, OOPSLA02*, 2002, pp. 119–130.
- [20] N. Glaser, "The CoMoMAS approach: From conceptual models to executable code," in *Proceedings of the 8th European Workshop On Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-97)*. Citeseer, 1997.
- [21] M. Wooldridge, N. Jennings, and D. Kinny, "The Gaia methodology for agent-oriented analysis and design," *Autonomous Agents and Multi-Agent Systems*, vol. 3, no. 3, pp. 285–312, 2000.
- [22] C. Sparkman, S. DeLoach, and A. Self, "Automated Derivation of Complex Agent Architectures from Analysis Specifications," in *Agent-oriented Software Engineering II: Second International Workshop*. Springer, 2002, p. 278.
- [23] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos, "Tropos: An agent-oriented software development methodology," *Autonomous Agents and Multi-Agent Systems*, vol. 8, no. 3, pp. 203–236, 2004.
- [24] S. Bussmann, N. Jennings, and M. Wooldridge, *Multiagent systems for manufacturing control: a design methodology*. Springer-Verlag New York, 2004.
- [25] L. Penserini, P. Bresciani, T. Kuflik, and P. Busetta, "Using Tropos to Model Agent Based Architectures for Adaptive Systems: A Case Study in Ambient Intelligence," in *Proceedings of the IEEE International Conference on Software Science, Technology & Engineering*. IEEE Computer Society Washington, DC, USA, 2005, pp. 37–46.
- [26] C. Iglesias, M. Garijo, and J. Centeno-González, "A Survey of Agent-Oriented Methodologies," in *Proceedings of the 5th International Workshop on Intelligent Agents*. Springer-Verlag London, UK, 1998, pp. 317–330.
- [27] C. Bernon, M. Gleizes, S. Peyruqueou, and G. Picard, "ADELFE: A Methodology for Adaptive Multi-agent Systems Engineering," in *ESAW 2002: revised papers*. Springer Verlag, 2003, p. 156.
- [28] C. Gershenson, "A General Methodology for Designing Self-Organizing Systems," *Arxiv preprint nlin/0505009*, 2005.
- [29] J. Fromm, "On engineering and emergence," *Arxiv preprint nlin/0601002*, 2006.
- [30] H. Turner, S. Stepney, and F. Polack, "Rule migration: Exploring a design framework for emergence," *International Journal of Unconventional Computing*, vol. 3, no. 1, p. 49, 2007.
- [31] T. De Wolf and T. Holvoet, "Towards a methodology for engineering self-organising emergent systems," *Self-Organization and Autonomic Informatics (I)*, vol. 135, no. 1, pp. 18–34, 2005.