

A generic software framework for role-based organic computing systems

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

Angaben zur Veröffentlichung / Publication details:

Nafz, Florian, Frank Ortmeier, Hella Seebach, Jan-Philipp Steghöfer, and Wolfgang Reif. 2009. "A generic software framework for role-based organic computing systems." In 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, 18-19 May 2009, Vancouver, BC, Canada, edited by Hausi A. Müller and Jeff Magee, 96-105. Piscataway, NJ: IEEE. <https://doi.org/10.1109/seams.2009.5069078>.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:

Deutsches Urheberrecht

Weitere Informationen finden Sie unter: / For more information see:

<https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/>



A generic software framework for role-based Organic Computing systems

Florian Nafz, Frank Ortmeier, Hella Seebach, Jan-Philipp Steghöfer and Wolfgang Reif
Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, D-86135 Augsburg
{nafz, ortmeier, seebach, steghoefer, reif}@informatik.uni-augsburg.de

Abstract

An Organic Computing system has the ability to autonomously (re-)organize and adapt itself. Such a system exhibits so called self-x properties (e.g. self-healing) and is therefore more dependable as e.g. some failures can be compensated. Furthermore, it is easier to maintain as it automatically configures itself and more convenient to use because of its automatic adaptation to new situations. Design and construction of Organic Computing systems are, however, challenging tasks. The Organic Design Pattern (ODP) is a design guideline to aid engineers in these tasks.

This paper introduces a generic software framework that allows for easy implementation of ODP-based Organic Computing Systems. The communication and service infrastructure of the multi-agent system Jadex is leveraged to provide interaction facilities and services to the application. The concepts of ODP are provided as generic, extensible elements that can be augmented with domain-specific behavior. The dynamic behavior of an ODP system is implemented and a generic observer/controller facility is provided. A real-world case study shows the applicability of the proposed approach and the handling of the software.

1 Introduction

The design of an Organic Computing (OC) system – i.e. a system that is able to self-organize, self-configure, self-heal, and self-adapt to changing environments or tasks – is still a complex issue. An important step on the way to a productive OC system is the ability to transfer the design artifacts to a software environment quickly and to have tools that support the development effort and the debugging of the application.

This paper proposes a technology that facilitates the implementation of systems that are based on the Organic Design Pattern (ODP) [15]. ODP enables a software engineer

to design OC systems in a fashion very similar to traditional software engineering approaches. With the help of the framework introduced in this paper, a developer can easily map the design artifacts of ODP to artifacts of a multi-agent system. This allows to construct the actual implementation of the system from the design documents, because many essential parts of an ODP application are already implemented in the generic software framework. The underlying foundation is provided by a mature multi-agent system (Jadex) with extensive tool support and documentation. As a consequence the implementation of an OC system can now be performed more easily and therefore much faster than before. Previous work [12] showed an implementation of the case study described in Section 5 with the multi-agent system AgentService [2]. This implementation was a domain-specific implementation, which was meant as a proof of concept and feasibility. The solution proposed now, however, enables the designer to easily transfer any system modeled in ODP to a software system.

The paper is organized as follows: Section 2 briefly introduces the Organic Design Pattern and its static and dynamic view. The foundation for the proposed framework – the multi-agent system Jadex – is introduced in Section 3. Afterwards, Section 4 illustrates the generic framework and the mapping of ODP design artifacts to artifacts of the software framework. Section 5 applies the principles to a case-study and shows the implementation of an actual application before Section 6 concludes the paper, highlights some related approaches and gives an outline of future work in the area.

2 Designing OC systems

The Organic Design Pattern (ODP) [15] is a design principle for a broad class of self-x systems, namely those which consist of a set of independent components interacting with each other and where self-healing, self-adaptation and other self-x properties can be achieved by a single mechanism:

computation of a new allocation of roles. It is possible to reason about the role allocations and prove certain properties about them.

Constructing a system with ODP is split into two phases: first, a domain-specific model has to be derived from the generic ODP (see Section 5.1) and second, the model has to be instantiated for a specific system (of this domain, see Section 5.3). This process corresponds to the software engineering tasks of capturing domain-specific concepts in classes and instantiating these classes according to a given situation.

The core of the pattern, an elaborate role concept, allows to model systems that adapt to changing tasks and can also process several resources with different tasks at the same time. Examples for such systems are sensor networks, distributed smart devices which provide context sensitive services, or adaptive production systems. Furthermore, the systems can run in a degraded mode in which a task is only partially fulfilled, thus compensating for failure as long as possible. The model has been based on a precise semantics which allows to define and measure self-x properties. Additionally, the reconfiguration process can be described on an abstract level very intuitively.

The following paragraphs give a very brief introduction to important concepts of ODP and the Restore-Invariant-Approach (RIA) for specification of reconfiguration algorithms. For more details see [15] and [8]. An application of this design process to a real-world case study will be shown in Section 5.

2.1 The Organic Design Pattern

An ODP system consists of *Agents* which process *Resources* with one or more of the agents' *Capabilities* according to a given *Task*. A *Task* describes how a given *Resource* should be processed. It is a sequence of *Capabilities* which should be applied to the *Resource*. The static view of such a system is shown in Figure 1.

Every *Agent* is characterized by the *Capabilities* it can provide and the agents to which/from which it can give/receive *Resources*. Which *Capability* an *Agent* performs in a specific situation is determined by its *Role*. An *Agent* can have several *Roles*. The mapping of *Roles* to *Agents* is called *role allocation* and is conceptualized in *allocatedRoles*.

Self-organization in this class of systems is described as a role allocation problem. A *Role* is a 3-tuple (*Precondition*, *Capabilities*, *Postcondition*) 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 how the resource is labeled and which agent should receive it. *Conditions* are 3-tuples of

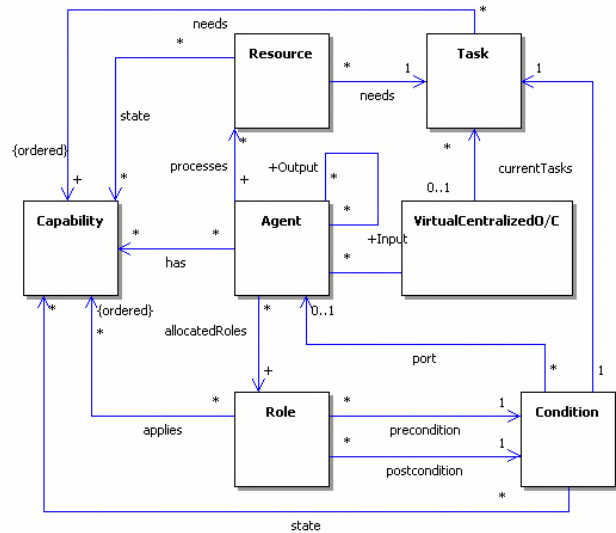


Figure 1. Organic Design Pattern

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. The reconfiguration mechanism is modeled in the concept of the *VirtualCentralizedO/C* (Observer/Controller [14], O/C). This component encapsulates the reconfiguration algorithm on design level. The output of the reconfiguration algorithm is a new allocation of *Roles* to *Agents* that restores the system to a state in which it is able to fulfill its tasks again.

2.2 Dynamic View

The dynamics of an ODP system are relatively simple. All *Agents* run asynchronously parallel. Dynamics can be split into two sub-domains: behavior during 'normal' (i.e. productive) phases and behavior when self-organization occurs.

During normal operation, interaction between *Agents* is done by passing *Resources* between *Agents*. Whenever an *Agent* receives a *Resource*, it chooses one of its allocated *Roles* according to the *precondition* and the *Capabilities* that have to be performed. Then the *Agent* applies the *Capabilities* defined in the chosen allocated *Role* to the *Resource*, refreshes the state and task of the *Resource* according to these *Capabilities* and gives the *Resource* to the *Agent* in the *postcondition* of the *Role*¹.

Reconfiguration is always triggered when a given role allocation is no longer correct. This can typically be mon-

¹Note, that the description above implies that the selection of a role generally depends on the *Resource* the *Agent* is actually processing. So there must also be a selection algorithm implemented, which chooses a *Role* if several allocated *Roles* are applicable at the same time.

itored during runtime by the agents. An example is that an agent loses a capability (maybe because of a hardware failure). The agent will then eventually receive a resource which cannot be processed with the remaining capabilities. This will trigger a reconfiguration². Whenever a reconfiguration is triggered, (1) all agents are informed to stop productive operation, (2) resources are cleared from the system, (3) data is gathered from the agents and (4) a new, valid role allocation is calculated and distributed.

2.3 Specification of Self-x Behavior

One of the major challenges in designing an ODP-based Organic Computing system is specification of the reconfiguration algorithm. In the context of this paper (and the reference implementation shown in Section 4), we restrict ourselves to sequential role execution where each agent performs only one role simultaneously³.

The big advantage of using the Organic Design Pattern is that it allows to systematically *design* self-x behavior by specifying properties for role allocations. These properties can be described as OCL constraints [13]. They can typically be split into two groups. The first group addresses consistency issues. Some examples are:

```

con1 inv: self.Input -> includesAll(self.
    allocatedRole.precondition.port)

con2 inv: self.Output -> includesAll(self.
    allocatedRole.postcondition.port)

con3 inv: self.allocatedRole.precondition.port
    -> forAll(b:Agent|b.allocatedRole.
    postcondition.port -> includes(self))

con4 inv: self.allocatedRole.postcondition.port
    -> forAll(b:Agent|b.allocatedRole.
    precondition.port -> includes(self))

```

Constraints 1 and 2 assert that an *Agent* can only receive resources from and give resources to *Agents* corresponding to its input and output associations and thus state that roles of an *Agent* have to be consistent with its input and output relations. Constraints 3 and 4 assert that if *Agent A* (referred to above as “self”) has *Agent B* as a port in the precondition of one of its *Roles*, then *Agent B* must have a role where *Agent A* is the port in the postcondition and vice versa. These kinds of constraints have to be considered when a new role-allocation is calculated. They determine the valid configurations for an ODP system and therefore the admissible results of a reconfiguration algorithm.

²In this context, it is assumed that an agent can detect the loss of a capability.

³Of course, each agent can have several roles at the same time. It just has to decide which of its roles it executes at any one time

The second group of constraints addresses properties, which must be monitored during runtime. They can very often (but not always) be decomposed and monitored by individual agents. The most important example is:

```

mon1 inv: self.has -> includesAll(self.
    allocatedRole.applies -> flatten())

```

This constraint asserts that the *Role* allocated to an *Agent* only includes *Capabilities* the particular *Agent* can perform. It must be monitored at runtime to ensure that an agent can perform the task it is supposed to do at any given time. If this constraint does not hold anymore, the system has to be reconfigured. As the constraint only uses data that is locally known to the agent, it can be monitored by the agent itself.

The constraints above are relatively generic. However it is possible that there exist additional, domain specific constraints (an example is shown in Section 5).

2.4 The Restore-Invariant-Approach

The constraints presented above are used at runtime to determine when a reconfiguration has to take place as well as during reconfiguration to ensure that the newly calculated role allocation fulfills the specification. Intuitively, the specification is fulfilled if the role allocation does not violate the OCL constraints. Whenever this statement of consistency does not hold, the system reconfigures to reestablish it.

An ODP system is defined through finite sets of agents, capabilities and resources. As described above, each agent has a number of roles that were allocated to it and a number of admissible inputs and outputs. Relations can be defined on these sets that determine the states the system can be in. The number of possible roles is also finite as duplicate roles are prohibited and the components of a role (pre- and postcondition, applies) are finite as well. The configuration of the system can be altered by assigning values to the free variables of the system – the *allocatedRoles* relation of each agent.

These properties have two positive consequences: (1) invariants can be monitored at runtime and – in most cases – locally by the agents and (2) correct reconfiguration can be stated as a SAT-solving problem (and therefore be solved by standard algorithms).

The Restore-Invariant-Approach distinguishes productive states in which the system performs its normal operations and reconfiguration states where a new role allocation is acquired and established. Whenever functional properties are not met (i.e. the system is not productive), a reconfiguration will be started which will reconfigure the system such that these properties are met again (i.e. it can be productive again). This intuitive notion of a self-x system is sufficient

to understand the remainder of this paper. For a more formal definition of the principles of RIA, please refer to [9] and [8].

3 The Multi-Agent System Jadex

The foundation of the proposed generic OC-framework is the multi-agent system (MAS) Jadex [4]. In development since 2002, Jadex is mature and has proven to be suitable for various academic and commercial applications.

Jadex 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.

Alternatively, Jadex can use the infrastructure provided by JADE [1], a multi-agent system whose main feature is strict adherence to the FIPA [6] specifications. These specifications provide interoperability between different MAS by providing a set of interfaces and an architecture a framework implementing the standard must adhere to. Most notably, this includes specifications for message formats and transportation. If required, Jadex can transparently use the structure and services provided by JADE without any change to the agents or their internal organization. When using JADE as the backbone, Jadex becomes fully FIPA compliant.

Jadex implements the notion of Belief, Desire, Intention (BDI) [3]. This paradigm has originally been developed to describe human perception and decision making and has been adopted by the agent community because it enables the description of reactive and proactive agents with intuitive terms. Additionally, BDI is a very flexible approach as it is not necessarily defined which actions will be taken in any given situation but the agents are capable of deliberating the next action based on their perceptions, their experience, and their goals.

The mental attitude of a Jadex agent is defined by its beliefs, its goals and its plans. A *belief* is an arbitrary object that holds any kind of information and is stored in the agent's *beliefbase*, 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 a very 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 a goal event is issued by the runtime environment. Such an event can be the trigger

for one or more plans. If there are several plans to handle a goal event, the Jadex meta-reasoning facility performs a deliberation to find and execute the most suitable one. In case the selected plan fails, the other plans are executed until the goal is fulfilled or no more plans are available. Furthermore, plans can be triggered by the arrival of messages from other agents (external events) or by internal events dispatched by plans of the same agent. 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 current *intention*. The plans contain the actions an agent is able to perform and are also referred to as *procedural recipes* as they are basically Java classes. A plan can instantiate (sub-)goals, send internal and external messages, and has the ability of synchronization with other agents by waiting for replies for messages or other plans within the same agent, e.g. by dispatching a goal and waiting for its success or failure, by waiting for a change in the beliefbase, or by waiting for an internal event.

In Jadex, agents can not be inherited from other agents. A very similar mechanism is however available with the notion of *capabilities*⁴. A capability can contain goals, plans, beliefs, events and all other parts of an agent definition but can not be instantiated directly. Instead, it can be imported by an agent and used in this container. All elements of a capability that will be used within an agent must be exported by the capability (thus providing a visibility mechanism just like providing private and public elements) and imported explicitly in the agent definition. As capabilities can import other capabilities, this mechanism can be used very flexibly to create hierarchies of capabilities.

Jadex' stability, its very thorough documentation and its acceptance in the agent community make it an ideal starting point for an implementation of a generic framework for role-based self-x-systems. The mapping of the concepts of ODP to Jadex concepts is outlined in the following section.

4 A Generic Foundation for OC Systems

ODP has been designed with multi-agent systems in mind from the beginning. It builds on the expertise and the structural principles of agent frameworks and enhances them with additional functionality to allow modeling and implementation of systems with self-x-capabilities. It is therefore possible to map the concepts of ODP to those of a sufficiently sophisticated multi-agent system such as Jadex. A large quantity of the pattern's static and dynamic structure can be implemented in a generic, reusable way. Therefore, it is possible to model a system in ODP and transfer it to the proposed generic framework with ease and without duplicating functionality common to a large variety of applications based on the pattern.

⁴Please note that this is quite different from the capabilities introduced in ODP!

ODP	Jadex	Remark
Agent	Agent	An ODP-Jadex agent imports the generic capability for basic ODP functionality.
O/C	Agent	The generic Observer/Controller handles the entire reconfiguration.
Capability	Plan	A capability that is applied to a resource must be implemented by the application developer. Within a role and the “has”-relation it is referred to by its name.
Role	Belief	A role consists of a precondition, a list of capabilities to apply and a postcondition.
Condition	Belief	Part of the role; it contains the resource state, task and an agent.
Resource	Belief	The resource is transmitted between agents as part of the resource exchange protocol.
Task		It contains the task field which describes what has to be done with the resource.
State		The state field indicates which steps have already been performed on the resource.

Table 1. Mapping of ODP concepts to Jadex concepts

As mentioned in Section 3, Jadex is based on artifacts which might be confusing from a more traditional software engineering point of view. As goals and plans are formulated explicitly, the business logic is split into two parts. One of them (goal) describes the state that the system wants to achieve, the other (plan) possible ways to get there. More often than not, there is only one viable way to achieve a certain goal, so the mapping between goal and plan becomes one to one. However, this fundamental principle still allows to formulate the functionality of a system in a very elegant way and provides a very simple yet powerful extension mechanism. The following sections first describe the implementation of a generic ODP agent and of the Observer/Controller as Jadex agents, both from a static and dynamic point of view and show how the concepts of ODP are mapped to Jadex concepts. Finally, the extension points a developer has to consider when implementing an actual application domain are presented.

4.1 The Generic ODP Agent

There is a lot of functionality that each ODP agent has to possess – regardless of the actual system it is part of – and which defines the execution semantics of the agent. Mainly, this is interaction with other agents, role selection, and handling of the reconfiguration process. These basic protocol and resource handling and reconfiguration mechanisms are implemented as the Jadex capability `OcInfrastructure` in order to provide them to the actual implementation of any specific type of ODP agent. The following paragraphs first describe the static view of the system and then show the dynamic execution semantics for production and reconfiguration phases.

Static View: The generic agent has a number of beliefs that correspond to the relations between the ODP concepts as described in Section 2.1. The capabilities the agent is able to apply at the moment are stored in the belief “has” which is basically a list of strings that contains string rep-

resentatives for every capability that can be applied. The relations “input” and “output” are expressed in a similar fashion. These beliefs are lists that contain the agent identifier for each agent the current agent can exchange resources with. The roles currently allocated to the agent are stored in the belief “allocatedRoles”. It contains a list of role descriptions which in turn contain a precondition, capabilities to apply (also encoded as strings) and a postcondition. The conditions are composed of an agent identifier (which describes from which agent a resource arrives or to which agent a resource should be given), the task that has to be performed on the resource and the state of resource processing. These conditions are evaluated whenever a role has to be selected or when the agent to which the resource has to be given is determined. There is also a belief for the resource that is processed by the agent (which defaults to null as long as the agent is not doing anything). The description of the resource also contains the relations for the state of the resource and its task. An overview of the static mapping between ODP concepts and Jadex concepts can be found in Table 1.

Dynamic View: As ODP is mainly designed for systems which process resources with a number of different agents, the generic goals of the ODP infrastructure mirror the resource flow paradigm. Taking a resource from another agent is triggered by the reception of the message *resource-available*. If an agent has received a resource which has to be processed, the *process-resource* goal is instantiated. Its aim is to apply the capability in the fitting role to the current resource. After this is done, *give-resource* is instantiated to trigger the transfer of the resource to the next agent in the chain according to the role that has also been used to process it.

There are two plans which handle the exchange of resources between agents: *take* and *give*. As mentioned before, there is no goal corresponding to *take* as the execution of the plan is triggered by the arrival of an external message. This message is actually generated as the first action of the

plan *give*. Whenever a resource has been processed, the goal *give-resource* is dispatched and *give* is executed in response. It sends the message *resource-available* to the agent set as the output in the role that has been used to process the resource. *Give* and *take* then engage in a short transmission protocol that is depicted in Figure 2. At the end of the conversation, a new resource is available to the taking agent and the *process-resource* goal is instantiated. The generic plan *choose-and-execute-role* is executed which tries to find a role that fits the status of the workpiece and instantiates goals which have the name of the capabilities in the “applies” relation of the role. The developer therefore has to provide a goal and at least one associated plan for each of the capabilities required for correct resource processing.

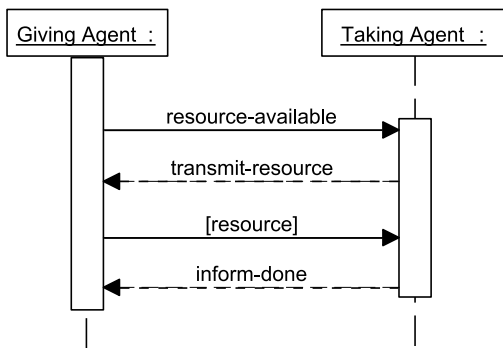


Figure 2. The protocol to exchange a resource between two agents

The reconfiguration of an agent is handled by another set of goals and plans. As a simplification, a reference to the observer/controller of the system is held at all times. A goal that contains the condition that a reference to the O/C is known exists. Whenever the *know-observer-controller* goal is triggered (i.e. whenever the agent has no reference to the observer/controller), the plan *search-observer-controller* is executed. It queries Jadex’ directory facilitator for an agent of type Observer/Controller and saves the reference to the O/C in the agent’s beliefbase, thus restoring the condition of the goal. If no O/C could be found the plan fails and is re-executed when the condition is checked the next time (after two seconds, the delay set in the goal definition).

More importantly, a goal that monitors the invariant (*monitor-invariant*) stating whether or not an agent is still functional is required (mon1, see Section 2.3). The goal contains a condition that expresses the requirement that all capabilities the agent has to apply according to its roles are available in the “has” relation. Additionally, there is an abstract goal *monitor-domain-invariants* that can be instantiated in an agent instantiation to enable the monitoring of domain-specific invariants. Whenever one of the conditions evaluates to false, an event is triggered and the plan *report-*

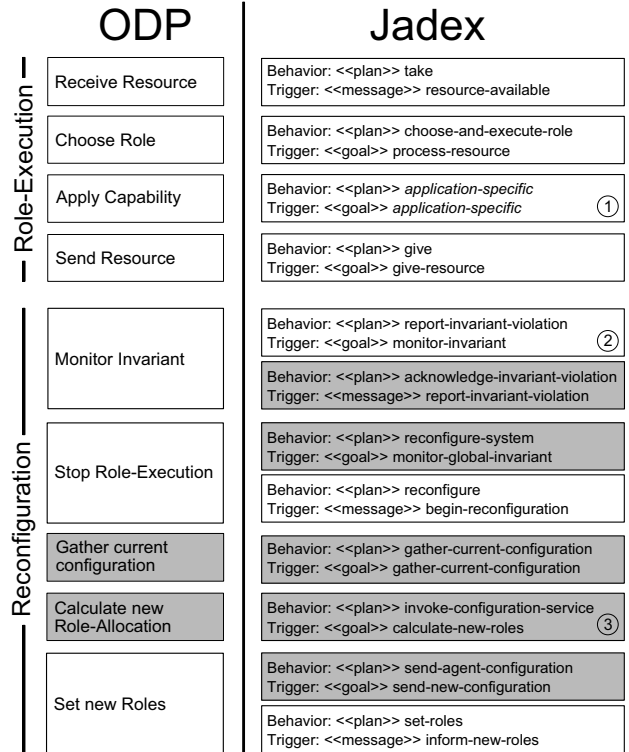


Figure 3. Mapping of the dynamic view of ODP

invariant-violation is executed to restore the invariants. It sets a flag in the agent’s beliefbase, indicating that the configuration is broken and that the agent is no longer running, reports the broken capability to the O/C and thus triggers the reconfiguration process. The O/C then sends out the message *begin-reconfiguration* to all agents which triggers execution of the *reconfigure-plan*. It first stops all ongoing resource processing, and then sends the current configuration of the agent, including all roles and the available capabilities to the O/C. As soon as the O/C calculated the new role allocation, it sends the *inform-new-roles* message, thus triggering the *set-roles* plan which puts the new roles into the beliefbase, informs the O/C of successful reconfiguration and waits for permission to begin operation again. As soon as the permission arrives, the flag indicating a bad configuration is reset and the agent is operational again. The mapping between the dynamic ODP view and the dynamic part of the Jadex implementation is depicted in Figure 3. White boxes indicate an action by the generic agent while gray boxes indicate actions by the observer/controller.

4.2 The Generic Observer/Controller

In the proposed framework, the main purpose of the generic observer/controller is handling of the reconfigura-

tion process, as the observation of correct functionality is already performed in a distributed fashion by the ODP agents.

The reconfiguration mechanism is triggered by an incoming *report-invariant-violation* message. A plan is executed that sets an internal flag, indicating the need to reconfigure the system. This flag is permanently observed by a goal which contains a condition to maintain the system functional⁵. Whenever the condition is violated, the *reconfigure-system* plan springs into action. It sends a message to all registered ODP-agents that a reconfiguration is in process, thus stopping the processing of resources and requesting their current configuration. At the same time, the *gather-global-knowledge* goal is dispatched which collects the incoming configuration from all agents and creates a global view on the current state of the system in the belief “agent_status”. As soon as all required data is available, a new goal – *calculate-new-configuration* – is dispatched. There are two different plans to achieve this goal at the moment, one which reads a new configuration from a file and a second one which transforms the configuration data into a format that can be transmitted over a web service. The web service is designed to act as a facade for an arbitrary, external reconfiguration algorithm. Whichever plan is used to calculate the new role allocation, as soon as the necessary data is available the O/C sends out messages containing the new roles for each agent. The agents acknowledge the new configuration and the O/C gives them permission to start operation again.

Additionally, the O/C provides roles during bootstrapping and maintains a list of ODP agents in the system. During system startup, all agents are searching for the O/C as described above. After they found it, they send an *agent-available* message. On reception, the O/C stores the agent identifier in its internal list of known agents (“agents”) and sends the agent its roles. These might be predefined roles or ones which have been calculated by the same means as during a reconfiguration. After the agent acknowledged the new configuration, the O/C sends a *begin-operation* message and the agent starts working.

4.3 Extension Points for Actual Implementations

The aim of a generic implementation framework is to make system construction easier and more efficient. A key requirement to achieve this goal is the definition of extension points which allow the developer to easily integrate domain- or application-specific logic. The most important extension points are outlined in the following and some of them will be illustrated in the implementation part of Sec-

⁵This indirection is introduced as the arrival of several *report-invariant-violation* messages could otherwise cause the O/C to start the reconfiguration process multiple times.

tion 5. The bracketed numbers refer to the corresponding numbers in Figure 3 and indicate which parts of the mapping are affected by the extension.

First, there are no generic goals or plans for the application of capabilities (1). Every domain implementation will work differently with the resources and the developer will have to accommodate this by defining goals and implementing the according plans. Depending on the number of different processing steps that have to be performed on a resource, there might be only one or several such goals and plans.

As mentioned before, each domain can contain domain-specific invariants that have to hold during runtime of a system of the domain (2). The generic ODP agent contains an abstract maintain-goal *monitor-domain-invariants* which can be instantiated in an agent implementation. The condition of the goal holds the domain-specific invariants and whenever the invariants are violated, the reconfiguration process is automatically started.

Furthermore, the algorithm that calculates the new role allocation during the reconfiguration will have to be adapted to the actual tasks present in the system (3). The framework provides a generic interface to a reconfiguration service based on the ODP concepts which can easily be reused in an application. However, calculation of the actual role allocation usually requires the incorporation of constraints defined specifically for a domain at design time.

Finally, if an agent does not adhere to the take-process-give schema of ODP resource processing, the developer will have to appropriately alter the plans such that the internal flow of the resource is handled differently. An example for this is presented with the concept of *Storages* in the case study.

5 Case study

The following case study illustrates the application of the proposed framework to a vision of future production automation systems. The example describes an autonomous production cell. Traditionally, production cells are designed as very static systems. Machines (for example robots) are linked in a strict sequential order (for example by conveyors). This results in very inflexible, rigid and error-prone systems where failure of any component will bring the whole system to a standstill.

The production automation domain presented in this paper consists of robots (each capable of using different tools), which are connected with autonomous transportation units and storages which provide and receive workpieces. The functional goal of the cell is to process workpieces following a given specification, the *task*. A task is a user-defined sequence of tool application.

The production cell used as an example contains three

robots, four autonomous carts and two storages (one for input and one for output). Each robot has three capabilities: drilling a hole in a workpiece, inserting a screw into a drilled hole and tightening an inserted screw. In one of the scenarios, every workpiece must be processed by all three tools in a given order (1st: drill, 2nd: insert, 3rd: tighten = DIT). Changing the tool of a robot requires a lot of time (compared to performing the task itself). Therefore the standard configuration of the system is to distribute the task among the robots, such that no robot has to switch tools and organize workpiece transportation accordingly. This situation is shown in Figure 4. The first robot drills a hole in the workpiece, the second one inserts a screw and the third one tightens this screw. Workpieces are provided and received by storages.

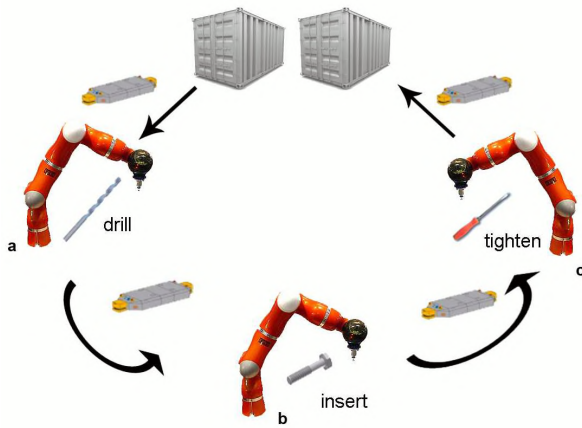


Figure 4. Adaptive production cell

5.1 Domain model

The first step of the modeling process is to instantiate the Organic Design Pattern and map its concepts to the domain. The result of the mapping is shown in Figure 5. The classes of the domain model have a link to the corresponding ODP concept in their upper right corner.

Static view: The production cell comprises three types of *Agents* – *Robots*, *AutonomousCarts*, and *Storages*. The capabilities of a robot are *Drill*, *Inserter* and *Screwdriver*, whereas an autonomous cart and a storage have no special capability. The fact that every robot has every tool adds redundancy and therefore a degree of freedom to the system. Due to the nature of the system, *Workpieces* (instances of *Resource*) can only be given from *Robots* or *Storages* to *Carts* and vice versa. This is captured by restricting *Input* and *Output* associations (so there is e.g. no *Input/Output* association between carts).

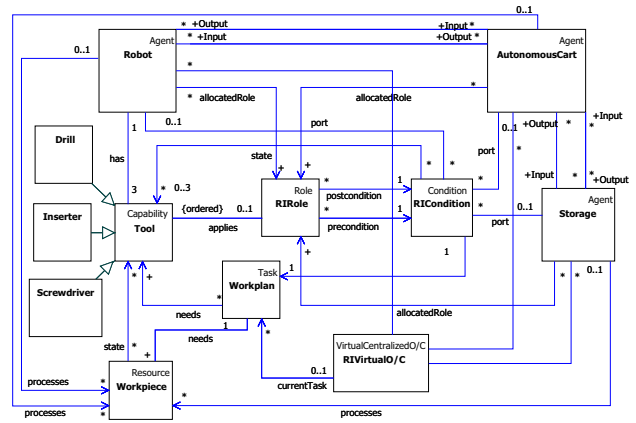


Figure 5. Domain model for adaptive production automation

The *Task* is a description of what has to be done. In the example: “first drill a hole (apply capability *Drill*), then insert a screw (apply capability *Inserter*) and finally tighten the screw (apply capability *Screwdriver*)”. A robot *Role* defines which capability a robot has to use, from which carts it is supposed to pick up workpieces and to which carts it should give the workpieces.

Reconfiguration: When a running system within this application domain has to be reconfigured, a new evaluation for the *allocatedRoles* relations has to be found that satisfies the consistency invariants as well as the locally monitored invariants. Section 2.3 described generic, domain-independent invariants which have to be considered for this. For the domain model described above, additional domain-specific invariants are necessary. For example, only *Storages* may introduce new workpieces or remove workpieces from the cell. This fact can be expressed in terms of the relations by stating that agents of type *Robot* and *Cart* can only have roles with non-empty precondition and postcondition. This is due to the fact, that a *Storage* produces a workpiece if it has a role with an empty precondition and a non-empty postcondition and receives and stores a workpiece if it has a role with a non-empty precondition and an empty post-condition. Such invariants have to be formulated in the domain model with OCL and can then be translated into a formal description that can be used in the reconfiguration algorithm.

5.2 Implementation

While the domain is being modeled, the domain concepts can be transferred to the proposed generic framework. The developer has to create agents according to the model

and provide plans for resource processing. Additionally, a reconfiguration algorithm has to be formulated, but this task is beyond the scope of this paper. The generic Observer/Controller can be used in this instance, so there is no need to adapt this component. The following paragraphs detail the creation of the agents and the instantiations of appropriate plans for each agent type.

The agent instantiations (Robot, Cart, and Storage) import the generic OcInfrastructure capability. This enables them to use the generic communication protocols and the reconfiguration facilities as well as the generic role-selection ability. The robot uses three additional Jadex capabilities, one for Drill, Insert, and Tighten respectively. These map directly to the ODP capabilities and each consist of a goal and a respective plan that is triggered by the goal. The goals are parameterized and take the resource to work on as a parameter. The plans use this parameter and alter the status of the resource by adding the capability name to the list of capabilities applied to the resource.

Carts and Storages work a little differently. As transportation and supply as well as reception of completely processed workpieces is not expressed as an explicit work step (i.e. there is no element in the list of capabilities to apply stating that a resource has to be transported, retrieved from a storage or put there), the role selection algorithm has to rely on implicit information. This is encoded in roles which have an empty “applies” field. For carts, the role contains an agent as the port in the precondition and an agent as the port in the postcondition. The cart transports resources between these two agents. Storages distinguish between two different types of roles: if a role has an empty precondition and an agent set as the port in the postcondition, a resource is retrieved from the store (i.e. a new resource object is created) and handed to the agent. If the precondition has an agent set as the port and the postcondition is empty, the Storage accepts a resource from the agent and “stores” it (i.e. the resource object is removed from the system). To use this implicit information, the original generic role selection mechanism has to be overwritten by an agent-specific one.

The way Storages handle the resources also means a derivation from the usual take-process-give routine of an ODP agent. First of all, a Storage that creates a new resource does not “take” it from another agent. Instead, resources are produced without a trigger from another agent. In the case study, the goal *create-resource* is therefore implemented as a maintain-goal that repeatedly checks if the Storage currently holds a resource in its beliefbase. If it does not (and there is an appropriate role as described above), a new workpiece is created and the *give-resource* goal is dispatched. Second of all, if a Storage is configured to receive and store a resource, the *give-resource* goal is never dispatched and therefore the workpiece remains at the Storage.

5.3 Instantiation

After refining the generic ODP to a domain-specific model, the next step is to instantiate this model to a specific system. This section shows this instantiation on the conceptual level and within the implementation framework.

Conceptual instantiation On a conceptual level, instantiation only means deriving an object-model from the domain-specific class diagram. For the discussed example the following object instances are created:

$$Ag := \{robot_a, robot_b, robot_c, cart_a, cart_b, cart_c, cart_d, storage_a, storage_b\}$$

$$Cap := \{d, i, t\}$$

where *d* (Drill), *i* (Insert) and *t* (Tighten) mean that *any* robot applies its respective tool.

Following this, the state of the system is defined by giving evaluations *val* for the relations. For initial system configuration, the associations *has*, *input* and *output* are defined as shown in Table 2.

Relation	Elements
$robot_x.has$	$\{d, i, t\}$
$robot_x.input$	$\{cart_a, cart_b, cart_c, cart_d\}$
$robot_x.output$	$\{cart_a, cart_b, cart_c, cart_d\}$
$cart_y.has$	$\{\}$
$cart_y.input$	$\{robot_a, robot_b, robot_c, store_a, store_b\}$
$cart_y.output$	$\{robot_a, robot_b, robot_c, store_a, store_b\}$
$storage_z.has$	$\{\}$
$storage_z.input$	$\{cart_a, cart_b, cart_c, cart_d\}$
$storage_z.output$	$\{cart_a, cart_b, cart_c, cart_d\}$
<i>currTask</i>	$\{[d, i, t]\}$
<i>currentAgents</i>	<i>Ag</i>

for $x \in \{a, b, c\}$ and $y \in \{a, b, c, d\}$ and $z \in \{a, b\}$

Table 2. Relations describing system state

During runtime, this object model typically changes, whenever (external) disturbances occur. For example a broken tool will result in a change of the *has* association, changes of available agents will result in changes of the agent set and new workpieces (with different tasks) will result in additional task objects. Note that invariants (i.e. the OCL constraints) are not further refined, because these describe expected behavior during reconfiguration. This is typically a domain-specific property and not a question of an actual instance of the system class.

JADEX instantiation An instantiation of the application is bootstrapped by a special manager agent. It creates the agents that are part of the instantiation of the domain according to a configuration file and shuts down afterwards. When the agents are started, the “input” and “output” relations are set as described above. The initial role allocation is provided by the Observer/Controller. Instantiation of a system thus becomes a mere matter of editing a configuration file.

6 Conclusion

This paper proposes a generic software framework for the construction of Organic Computing systems. The framework provides most functionality required to implement an Organic Computing system based on the Organic Design Pattern (ODP). ODP is a very useful guideline and design process for the efficient construction of OC systems. The proposed framework closes the gap between design, modeling and the actual implementation of Organic Computing systems.

The implementation of the framework is based on the multi-agent system Jadex. While the idea of using multi-agent systems in the context of Organic Computing applications is not new (see for example [11, 10, 5]), none of these approaches aims at building an implementation *framework*. All present approaches we know of focus only on using a multi-agent systems for implementation of a given target system. In contrast, the approach proposed here tries to enhance a state-of-the-art multi-agent system such that Organic Computing can be easily and efficiently realized on this basis.

One approach that is similar to the design principle ODP is presented in [7]. Architectural constraints, i.e. rules that limit the way components may be composed, are used to impose the specification of the system on instances that consist of autonomous components which self-organize their structure. It would be very interesting to see if the proposed software framework can be adapted to this modeling paradigm.

Our future work will focus on dealing with extensions to the class of target systems (i.e. a planned generalization of the Organic Design Pattern). Furthermore, the presented case study will be used to evaluate different reconfiguration algorithms (e.g. of-the-shelf constraint solvers in comparison to genetic algorithms) and the connection to a cyber-physical simulation environment. Both tasks will significantly benefit from the generic Jadex-based implementation framework proposed in this paper.

References

[1] F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with a FIPA-compliant agent framework. *Soft-*

ware Practice and Experience, 31(2):103–128, 2001.

[2] A. Boccalatte, A. Grosso, and C. Vecchiola. Implementing a mobile agent infrastructure on the .NET framework. In *4th International Conference in Central Europe on .NET Technologies*, May 2006.

[3] M. Bratman. *Intention, plans, and practical reason*. Harvard University Press, 1987.

[4] L. Braubach, A. Pokahr, and W. Lamersdorf. Jadex: A BDI Agent System Combining Middleware and Reasoning. *Software Agent-Based Applications, Platforms and Development Kits*, 2005.

[5] D. Fey, M. Komann, F. Schurz, and A. Loos. An Organic Computing architecture for visual microprocessors based on Marching Pixels. In *IEEE International Symposium on Circuits and Systems, 2007. ISCAS 2007*, pages 2686–2689, 2007.

[6] FIPA, Foundation for Intelligent Physical Agents, 1996. <http://www.fipa.org/>.

[7] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 33–38, New York, NY, USA, 2002. ACM.

[8] M. Güdemann, F. Nafz, F. Ortmeier, H. Seebach, and W. Reif. A specification and construction paradigm for organic computing systems. In S. Brueckner, P. Robertson, and U. Bellur, editors, *Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 233–242. IEEE Computer Society Press (2008), 2008.

[9] M. Güdemann, F. Ortmeier, and W. Reif. Formal modeling and verification of systems with self-x properties. In *Proceedings of the Third International Conference on Autonomic and Trusted Computing (ATC-06)*, volume 4158 of *Lecture Notes in Computer Science*, pages 38–47, Berlin/Heidelberg, Sept. 2006. Springer.

[10] H. Kasinger and B. Bauer. Combining Multi-Agent-System Methodologies for Organic Computing Systems. In *Database and Expert Systems Applications, 2005. Proceedings. Sixteenth International Workshop on*, pages 160–164, 2005.

[11] H. Kasinger, B. Bauer, and J. Denzinger. The meaning of semiochemicals to the design of self-organizing systems. In *SASO '08: Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 139–148, Washington, DC, USA, 2008. IEEE Computer Society.

[12] F. Nafz, F. Ortmeier, H. Seebach, J.-P. Steghöfer, and W. Reif. Implementing Organic Systems with AgentService. In *Evaluation of Novel Approaches to Software Engineering ENASE 2008*. Springer, 2008.

[13] OMG. Object Constraint Language, OMG Available Specification, 2006.

[14] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer, and H. Schmeck. Towards a generic observer/controller architecture for Organic Computing. *INFORMATIK*, pages 112–119, 2006.

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