OPEN ACCESS
OPUS AUGSBURG

Universität Augsburg
Universitätsbibliothek

A specification and construction paradigm for organic computing systems

Matthias Güdemann, Florian Nafz, Frank Ortmeier, Hella Seebach, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Güdemann, Matthias, Florian Nafz, Frank Ortmeier, Hella Seebach, and Wolfgang Reif. 2008. "A specification and construction paradigm for organic computing systems." In Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, 20-24 October 2008, Venezia, Italy, edited by Sven Brueckner, Paul Robertson, and Umesh Bellur, 233–42. Piscataway, NJ: IEEE. https://doi.org/10.1109/saso.2008.66.

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 Specification and Construction Paradigm for Organic Computing Systems

M. Güdemann, F. Nafz, F. Ortmeier, H. Seebach and W. Reif
Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, D-86135 Augsburg
{guedemann, nafz, ortmeier, seebach, reif}@informatik.uni-augsburg.de

Abstract

Organic Computing systems are systems which have the capability to autonomously (re-)organize and adapt themselves. The benefit of such systems with self-x properties is that they are more dependable, as they can compensate for some failures. They are easier to maintain, because they can automatically configure themselves and are more convenient to use because of automatic adaptation to new situations. While Organic Computing systems have a lot of desired properties, there still exists only little knowledge on how they can be designed and built.

In this paper an approach for specification and construction of a class of Organic Computing systems is presented, called the Restore Invariant Approach *(RIA). The core idea is that the behaviour of an Organic Computing system can be split into productive phases and self-x phases. This allows for a generic description of how "organic" aspects can be specified and implemented. The approach will be illustrated by applying it to a design methodology for Organic Computing systems and further refining it to an explicit case study in the domain of production automation.*

Key words: formal methods, safety critical systems, organic computing, autonomic computing

1 Introduction

There exist a lot of different notions of what Organic Computing (OC) [12] and Autonomic Computing (AC) [9] systems are and what properties they are supposed to fulfill. One large class of such systems are those with self-x capabilities.

Self-x systems are characterized by being able to autonomously adjust to different situations. For example a self-configuring system is expected to dynamically configure new components, a self-adapting system is expected to autonomously react on changing requirements and a self-healing system is expected to be able to compensate component failures. In addition to these self-x properties, functional requirements are to be met as well, i.e. the system is expected to produce, calculate or measure something whenever no adaptation is necessary.

While such self-x properties are desired in a wide variety of application areas, it still remains a major challenge how such systems can be built or even be specified. There exist a lot of domain specific approaches [19, 1, 17, 18], which are not general enough to be applicable in other domains.

In this paper we present a generic approach for specifying and constructing a specific class of Organic Computing systems, namely agent and role based systems. It basically relies on the following informal notion of what a self-x system is expected to do: *"Whenever functional properties are not met (i.e. the system is not productive), then a self-x phase will be started which will reconfigure the system such that these properties are met again (i.e. it can be productive again)."* A formalization of this intuitive statement can be used for specification and construction of Organic Computing systems. The core idea is to use invariants to specify productive states (resp. self-x states) and to use constraint solving techniques, e.g. [8, 23], (whenever the system is not operational) to find new configurations such that the system can become operational again.

A brief overview of a specific class of Organic Computing systems and a generic modeling framework is given in Sect. 2. Sect. 3 explains the idea of using invariants for specification of organic behavior. An application of this method to a case study from production automation is shown in Sect. 4. Other approaches and future work are discussed in Sect. 5. Concluding remarks are summarized in Sect. 6.

2 Self-x Systems

Most Organic Computing systems consist of two parts: one which provides the basic functionality (productive part) and the organic part which is responsible for (re-) configuration and adaptation of the system. The productive part is directly related to the intention of the system. Examples are data acquisition, data relaying or data aggregation in the context of sensor networks[17]. Another example is the control and driver logic for operating traffic lights in a city wide traffic control system [16].

The organic part normally consists of some kind of observation and some kind of control algorithm. In the examples above, observation is detection of sensor failures or measurement/prediction of major changes in the traffic flow. In the sensor network scenario control is the calculation of new routing for data or determination of new time intervals for green and red phases for the traffic lights.

Technically, the organic part and the productive part of the system are often connected with an self-x infrastructure. The self-x infrastructure wraps components of the productive part of the system in a generic interface. It allows for reconfiguration of these parts as well as for information exchange. The organic part constantly monitors the functional part and uses control whenever necessary.

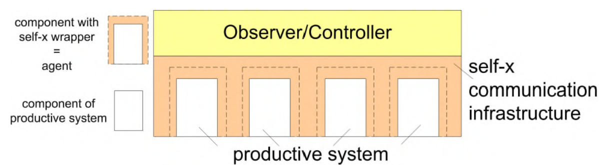

Figure 1. Architecture for Organic Computing systems

Figure 1 shows such an architecture with one (central) observer/controller. Decentralized versions and hierarchical observer controllers are also often used for organic control mechanisms [18]. Regardless of the type of observer/controller architecture being used, models of Organic Computing have very similar semantics.

Semantics In most cases, the semantics of Organic Computing systems is a trace semantics. This means a system model describes a set of traces, where each trace is a sequence of states. A state then describes a snapshot of the system at some given point in time. One sequence describes one possible temporal evolution of the system over time – also called a run or trace of the system. The semantics of the system is then the set of all possible runs. This type of semantics is very widely used in computer science for all types of systems [15, 3, 7]. There exist a lot of different specification formalisms the rely on a trace semantics, e.g. programs, state charts and sequence charts.

Independent of the selected specification formalism, Organic Computing systems share one common property: it is possible to decide at a given time if the system is providing functionality, i.e. it is in a productive phase, or if it is adapting to changes in the environment, i.e. it is in a self-x phase. Therefore all states can be divided into two disjoint sets:

- one set of states where the functionality can be provided and no reconfiguration is necessary
- one set of states where the functionality can not be provided and where reconfiguration is or must be performed

Formally, this splitting of system states can be expressed by a predicate logic formula, which evaluates to true for all states of the first group and to false for all others. As the formula is true for all functional states, we will speak in the following of an invariant[1].

This leads to the core idea of this paper: a generic paradigm for specification and construction of Organic Computing systems. Assume that the initial state of the system is part of the functional states. Then the following statement describes the essence of an Organic Computing system with the notion of invariants:

> *An Organic Computing system behaves like a standard system as long as no reconfiguration is required. This situation corresponds to a satisfied invariant. Adaptation is necessary, if the invariant does not hold anymore. An Organic Computing system is expected to restore its functionality (or invariant) as far as possible.*

In simple words: building an Organic Computing system can be reduced to building the functional part of the system, wrapping it into a self-x infrastructure, defining invariants and providing an algorithm which can restore these invariants. The good news is that defining suitable invariants is often possible in a very generic way and that the restoration problem is basically a constraint solving problem.

This very abstract approach can be applied to a broad variety of Organic Computing systems. In the following section, we will show how it works for a specific class of Organic Computing systems.

[1]This is justified, as the formula basically is a formalization of the goals of the system. For a traditional (non-organic) systems, such properties are typically called invariants as they are expected to be satisfied by all states of every run of the system.

3 Restore Invariant Approach

This section describes a generic formalism for specification and implementation of Organic Computing systems or more specifically for specifying and implementing the organic intelligence of such systems.

The first part of this section describes a specific class of Organic Computing systems and proposes a formalism to model such systems. This formalism is based on [20] and will be extended in this paper by the *Restore Invariant Approach* . The second part of the section will show how invariants, as presented into the previous section, for this class of target systems look like. The last part of the section gives some remarks on one implementation for restoration of the invariants.

3.1 Static View

The Organic Design Pattern (ODP)[20] is a design and construction guideline for a broad class of self-x systems, namely those which consist of a set of independent components interacting with each other and where reconfiguration/adaptation can be expressed as a reallocation of roles. For that purpose the components must provide several functionalities instead of only one. Some examples for such systems are sensor networks, distributed smart devices which provide context sensitive services, or adaptive production systems. An example for this is shown in Sect. 4. Fig. 2 shows the Organic Design Pattern (ODP).

The main idea is that a system consist 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*. Every *Agent* is defined 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*. A *Role* is a 3-tuple *(Preconditions, Caps, Postconditions)* of a set of *Conditions* as precondition, a sequence of *Capabilities* that need to be applied and a set of *Conditions* as postcondition. The precondition describes which resources are to be accepted and from whom. The postcondition describes how the resource is to be labeled and to whom it is given[2]. A *RoleAllocation* is the set of all *Roles* of all *Agents*. In the following, we restrict ourself to a single precondition and a single postcondition[3]. A *Condition* is 3-tuple of a target *Agent* from whom respectively to whom the *Resource* is taken respectively given, the current *State* of the *Resource* and the *Task* that needs to be done. An Agent chooses a *Role* according to the *Precondition*, performs the *Capabilities* defined in the *Role* and give the *Resource* to the Agent in the *Postcondition*. Note, that the definition above implies that the selection of a role is in general dependant on the *Resource* which the agent is actually processing.

3.2 Dynamic View

The core part of the semantics which is necessary for the description of invariants as basis for reconfiguration, is how roles determine the system behavior. The intention of the discussed class of target systems is to process resources in a given way, e.g. transporting some information from a sensor node to a data sink in a sensor network or to enrich, aggregate and prepare some information from different clients for the user. This is done by each agent individually by following three rules:

1. Accept resources only, if the tuple (sending agent, current state of the resource, task of the resource) matches one of the roles preconditions.
2. Process resources according to the capabilities defined in the chosen role.
3. Try to give resources to the next agent as stated in the postcondition of the selected role. State and task are also changed as stated in the postcondition[4].

Formally, this corresponds to the idea of a data flow network, where resources are the data, agents are the nodes of the network and the edges are defined by the roles assigned to the agents. We also used such a formalization to model the example of Sect. 4 in the data flow based tool SCADE [6].

We do not give a complete semantics of the dynamics of the system here. The idea is to use state charts with a defined semantics for this. In this paper we abstract from explicit communication protocols like for example handshakes and also from properties dependent on the chosen protocol, e.g. avoiding deadlocks.

3.3 Invariant

This subsection will explain the invariants in more detail. As mentioned before, the static artifacts *Agent*, *Capability*, *Task* and the association between them are enough to specify if a role allocation is correct or not. This is also intuitively justified, as the configurations of the agents including

[2] Note, that in general there must also be a decision algorithm implemented, which chooses a role if several roles are applicable at the same time.

[3] The opportunity to define more than one pre- and/or postcondition in one role serves for merging and splitting resources and permits optimization e.g. load balancing.

[4] In almost all applications the state is updated by adding the capabilities which have been applied and leaving the task unchanged. Different scenarios are only interesting for applications where resources are being split into pieces or merged together.

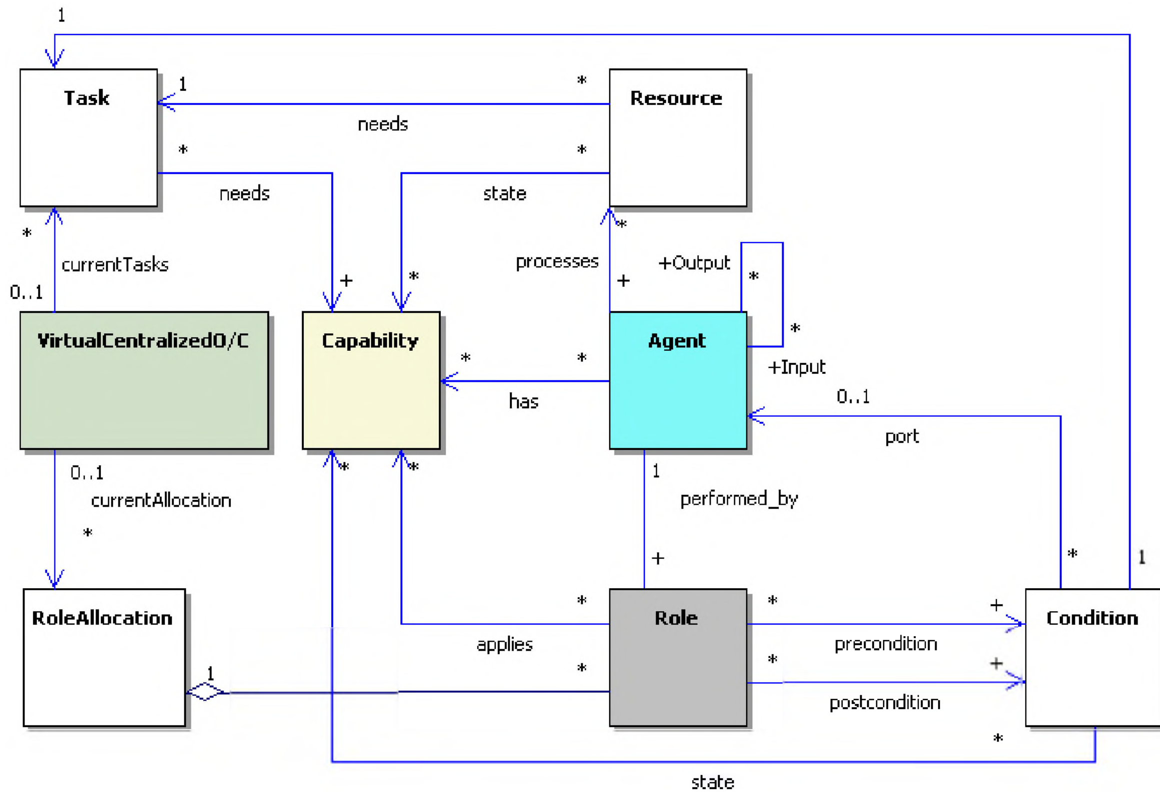

Figure 2. Organic design pattern (ODP)

their roles solely determines whether the system will work correctly or not[5]. The next step is to define a formula (or invariant), which divides role allocations in "good roles", where the functionality can be provided, and "bad roles", where the functionality cannot be provide and therefore reconfiguration is necessary.

In Fig. 3 an algebraic formalization of the class diagram of Fig. 2 is given. It is based on the standard semantics of UML class diagrams.

A specific system is described by a (finite) set of variables $Agents_{all}$ of type *Agent* and $Tasks_{all}$ of type $Task$. The recursive data type *Agent* is a 4-tuple of a list of *Capabilities*, a set of (input) *Agents*, a set of (output) *Agents* and a set of assigned *Roles*. A *Role* is a 3-tuple of (pre-) *Conditions*, (post-) *Conditions* and a list of *Capabilities*, which are to be applied. *Conditions* are a 3-tuple of a (target/source) *Agent* (called port), a list of *Capabilities* describing the current state of the resource and a list of *Capabilities* describing the task which should be conducted on the resource. These variable declarations define the state space (for the abstraction used for specification of the invariants) for all possible systems within the class of systems defined by the ODP. The data type *Capability* is left abstract on this generic level. On the application level it is updated by the concrete type.

[5] As we assume that the dynamics of the system are implemented correctly with respect to the informal semantics given in Sect. 3.2.

For example a system with four agents and one task will lead to the set $Agents_{all} := \{a_1, a_2, a_3, a_4\}$ and $Tasks_{all} := \{t_1\}$. For a full specification the variables $Agent.inputs$, $Agent.outputs$, $Agent.has$ (for $Agent = a_{1..4}$) and $Task$ (for $Task = t_1$) have to be defined. All other variables remain free. All possible configurations are now defined by the possible evaluations of the remaining free variables.

In general the invariant can be split into two types of subformulas: consistency predicates and configuration predicates.

Consistency predicates: Consistency predicates $\mathit{INV}_{\text{cons}}$ express that roles are consistent with the associations between the static artifacts and are derived from the OCL constraints which can be annotated to the design pattern. During runtime these parts of the invariant must be monitored.

An example is that only capabilities can be assigned

type Agent{

has :	set of Capability
inputs :	set of Agent
outputs :	set of Agent
assignedRoles :	set of Role

}

abstract type Capability{ }

type Role{

precondition :	set of Condition
applyCapabilities :	list of Capability
postconditions :	set of Condition

}

type Condition{

port :	Agent
state :	list of Capability
task :	Task

}

type Task{

capSeq :	list of Capability

}

Figure 3. Formal representation of ODP artifacts

within a role to an agent if this agent has this capability. In formal language:

$$\forall a \in Agent_{all}: \forall r \in a.assignedRole: \\ \forall c \in r.applyCapabilities: c \in a.has$$

The set of available capabilities ($a.has$) may change whenever failures occur. In reality this corresponds to a situation, in which for example a tool of a robot breaks (see Sect. 4).

Another example is, that if agent A wants to give resources to agent B then agent B must have the possibility to receive resources from agent A.

Another predicate in this group is, that ports of a role of an agent have to be consistent with the agent's input/output relation.

$$\forall a \in Agent_{all} : \forall r \in a.assignedRole: \\ (r.precondition.port \in a.input) \\ \wedge (r.postcondition.port \in a.output)$$

These predicates are monitored by the system during runtime. Failures lead to a violation of one of those predicates, and a reconfiguration is triggered.

Configuration predicates: Configuration predicates INV_{conf} express correct calculation of role allocations. They describe functional properties of the system and (normally) do not change during runtime. An example is that the roles should be allocated in a way that resources are processed correctly, i.e. correct order of applied capabilities. Therefore one of the Configuration predicates of the invariant is that agents which need to exchange resources need to be connected. In other words the ports set in the conditions must be connected in such a way that if one agent should give resources to another, this one must have this agent set as an agent it gets resources from.

$$\forall a, a' \in Agent_{all} : \forall r \in a.assignedRole : \\ a' \in r.postcondition.port \\ \rightarrow \exists r' \in a'.assignedRole : (a \in r'.precondition.port)$$

Two other Configuration predicates are stating that the state of resources is updated correctly (1) or that the already applied capabilities are a prefix of the list of capabilities in the task (2).

$$(1)\ \forall r : Role : \\ r.postcondition.state = \\ r.precondition.state{+}{+}r.applyCapabilities$$

$$(2)\ \forall c \in Condition : c.state \sqsubseteq c.task$$

Some other necessary predicates are specified analogously. For example: there exists at least one agent that finishes the task or that there is always the next capability of the task applied.

3.4 Restoration of Invariant

The last paragraph showed, how invariants can be used for specifying organic behavior in a very generic and intuitive way. The next logical step is to define a reconfiguration algorithm on top of this. A fairly simple way to accomplish this is to use a constraint solver.

Formally the invariant is a boolean formula $INV(vars_{stat}, vars_{dyn})$ where $vars_{stat}$ represent variables that can not be changed by the controller, for example the input relation $Agent.inputs$ or the available capabilities $Agent.has$, and where $vars_{dyn}$ represents variables that can be changed by the controller, for example allocated roles $Agent.assignedRoles$. The goal of the Organic Computing system is to keep *INV* true as long as possible. So the question of reconfiguration can be reduced to the following logic problem:

Given an invariant $INV(vars_{stat}, vars_{dyn})$ and an evaluation $\pi_{stat} : vars_{stat} \rightarrow Dom(vars_{stat})$[6] find an eval-

[6] Here $Dom(vars_{stat})$ denotes the vector of domains which correspond to the vector of static variables $vars_{stat}$. Analogously for dynamic variables.

uation π_{dyn} : $vars_{dyn} \rightarrow Dom(vars_{dyn})$ such that $INV(vars_{stat}, vars_{dyn})_{|\pi_{stat} \cup \pi_{dyn}} = tt$[7].

More intuitively one can say, that a valid evaluation of variables for a induced formula $INV'(vars_{dyn})$ must be found. This is a standard problem of constraint solving. Therefore the problem of designing an organic reconfiguration algorithm can be reduced to specifying invariants of intended functional properties, defining degrees of freedom and then using an "of-the-shelf" constraint solver. How this works in practice will be described in the next section.

4 Case Study

This section describes an application from production automation. The first part of this section describes the system and how it is modeled. The second part shows some experiences and results.

4.1 Adaptive Production Cell

In contrast to a traditional production cell where the interaction between robots is fixed, the adaptive production cell will dynamically change its interaction scheme. In the example, the production cell consists of three robots, which are connected with two autonomous transportation units. This is shown in Fig. 4.

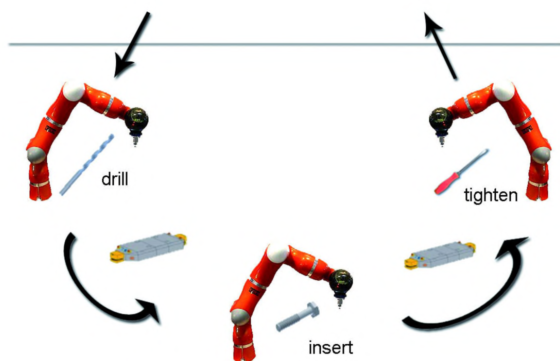

Figure 4. Adaptive production cell

Their task is to drill a hole, then to insert a screw and in a final step to tighten the screw. For the modelling process the first step is to instantiate the ODP and identify the agents. In Fig. 5 an extract of the model of such a system is given in ODP notion, which focuses on the instantiation of *Agent* and *Capability*. A more detailed description of the instantiation of the ODP is given in [20]. For better readability, capabilities are represented as data types, while robots and carts are object representations.

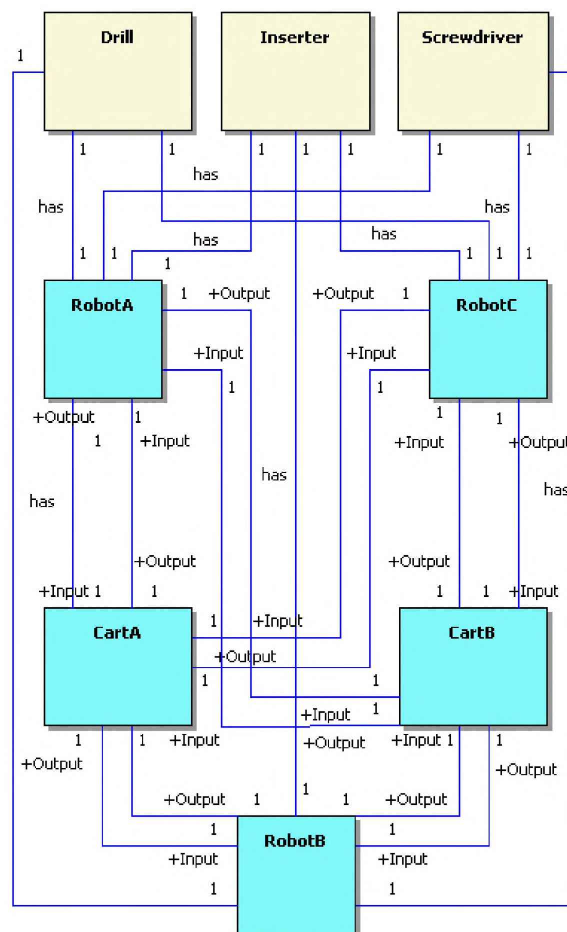

Figure 5. Extract of the object model for the adaptive production cell

The production cell comprises two types of *Agents* - *Robots* and *AutonomousCarts*. The capabilities of a robot are *Drill*, *Insert* and *Tighten*, whereas an autonomous cart has no special capability. Fig. 5 shows the optimal case in which the system has full capabilities and functionality.

Due to the nature of the system *Workpieces* (instances of *Resource*) can only be given from *Robots* to *Carts* or vice versa. This is captured by restricting *Input* and *Output* associations.

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 *Insert*) and finally tighten the screw (apply capability *Tighten*)". A robot role defines,

[7] Here $\pi_{stat} \cup \pi_{dyn}$ denotes the evaluation π of $vars_{stat} \cup vars_{dyn}$ which is defined by applying π_{stat} and π_{dyn}.

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.

This system is now modeled, the invariants and reconfiguration are specified according to the method described in Sect. 3. The set of instances of *Agent* is

$$Agent_{all} := \{robot_a, robot_b, robot_c, cart_a, cart_b\}.$$

The set of capabilities and all task is defined analogously

$$Capability_{all} := \{d, i, t\} \qquad Tasks_{all} := \{t_1\}$$

Next we need to define the state of the system. This is done by defining the evaluations π_{stat} for all variables of $vars_{stat}$.

In table 4.1 the static variables and their evaluation are given for the example according to the definitions of Sect. 3. The evaluation describes the initial situation of the production cell in which every agent has full capabilities and every robot can interact with every cart and vice versa. This situation is also depicted in Fig. 4.

The functional goal of the cell is to process workpieces following a given work plan. In the example the work plan should be: drill a hole, insert a screw and tighten the screw, this is described by setting the field $t_1.capSeq$ to the according value.

$\mathbf{vars_{stat}}$	$\pi_{\mathbf{stat}}(\mathbf{vars_{stat}})$
$robot_x.has$	$\{d, i, t\}$
$robot_x.inputs$	$\{cart_a, cart_b\}$
$robot_x.outputs$	$\{cart_a, cart_b\}$
$cart_y.has$	$\{\}$
$cart_y.inputs$	$\{robot_a, robot_b\}$
$cart_y.outputs$	$\{robot_a, robot_b\}$
$task_1.capSeq$	$[d, i, t]$

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

Table 1. Evaluation π_{stat} of $vars_{stat}$

As described in Sect. 3 reconfiguration is done by finding evaluations of the dynamic variables $vars_{dyn}$, such that the invariant *INV* holds.

Table 4.1 shows a possible solution for the system which is represented by the evaluation of the dynamic variables $vars_{dyn}$. For simplification, we only show the solution for one robot ($robot_a$) and one cart ($cart_b$). The evaluation for the other agents looks similar.

During runtime the consistency part of the invariant INV_{cons} is constantly monitored. this is typically achieved by integrating a sensor, which can for example detect if a specific capability can be accessed or not.

$\mathbf{vars}_{dyn}$	$\pi_{dyn}(\mathbf{vars}_{dyn})$
$robot_a$.*assignedRoles.precondition.port*	$\{cart_a\}$
$robot_a$.*assignedRoles.precondition.state*	{*d*}
$robot_a$.*assignedRoles.precondition.task*	{*d,i,t*}
$robot_a$.*assignedRoles.applyCapabilities*	{*i*}
$robot_a$.*assignedRoles.postcondition.port*	$\{cart_b\}$
$robot_a$.*assignedRoles.postcondition.state*	{*d,i*}
$robot_a$.*assignedRoles.postcondition.task*	{*d,i,t*}
$cart_b$.*assignedRoles.precondition.port*	$\{robot_a\}$
$cart_b$.*assignedRoles.precondition.state*	{*d,i*}
$cart_b$.*assignedRoles.precondition.task*	{*d,i,t*}
$cart_b$.*assignedRoles.applyCapabilities*	{}
$cart_b$.*assignedRoles.postcondition.port*	$\{robot_c\}$
$cart_b$.*assignedRoles.postcondition.state*	{*d,i*}
$cart_b$.*assignedRoles.postcondition.task*	{*d,i,t*}

Table 2. Evaluation for $robot_a$ and $cart_b$

Whenever a broken tool is detected, the corresponding (static) variable $a.has$ will be assigned a new value. As a consequence the invariant will become false, if the failure has some effect on the functionality. If now, for instance, the drill of $robot_a$ breaks, then the evaluation of $robot_a.has$ will be changed from $\{d, i, t\}$ to $\{i, t\}$. As $robot_a$ is configured to insert (and not to drill) this loss of capabilities does not affect the system and thus no reconfiguration will be triggered. Note, that the consistency predicate

$$\forall a \in Agent_{all}\colon \forall r \in a.assignedRole: \\ \forall c \in r.applyCapabilities: c \in a.has$$

is still evaluated to true, although the value of $robot_a.has$ has changed. Now assume that also the tool breaks, which $robot_a$ uses for inserting screws. As a consequence evaluation of $robot_a.has$ will be changed to $\{t\}$. This has now some consequence on the invariant. The formula above will now evaluate to false (for $a := robot_a$). therefore a reconfiguration will be triggered. The observer/controller is now trying to find a new evaluation of the dynamic variables with respect to the new evaluation of $vars_{stat}$, which restores the invariant. One obvious solution is to switch the roles of the inserting and the tightening robot and reassign the carts accordingly. This is then called self-healing: the system became non-functional and re-organized itself in such a way that it became functional again.

Another example is self-adaptation to a new task. For example "Drill three holes" is reflected in the change of the evaluation of the task variable t_1.

$$t_1.capSeq = [d, d, d]$$

The reconfiguration then is calculating new assignments, so that the new task is fulfilled.

4.2 Implementation and Experiences

We implemented this case study using *Microsoft Robotics Studio* as simulation environment, the multi-agent framework *JADEX* as communication infrastructure and the constraint solver *Alloy* for restoring invariants. Adapters between JADEX objects and Alloy syntax were relatively easy to implement. A video of the system can be found at [13]. The definition of the invariants was also relatively straight forward. For a slightly larger system consisting of three robots, five carts and two storages, the constraint solving problem only takes a couple of seconds. Although we have not yet proven the proposed set of invariants as correct for the whole class of target systems, we were at least unable to find counter-examples.

However we experienced, that in reality there often exist hidden links between the individual components. An example is, that assigning roles to carts might cause the carts to collide while travelling from one robot to another. The reason for that is that the topology of the system is not considered in the role assignment. This seems to be possible for many systems where the (software) agents represent physical entities. But solutions for this problem are only solvable in an application-specific way.

The next step in our research is to implement a distributed algorithm for reconfiguration. As for now, the observer/controller is a single-point-of-failure for the system. A simple extension is to add this logic to all agents and do a leader election, whenever a re-configuration is necessary. Further extension might include applying local reconfigurations, where only a limited number of agents exchange roles.

As the state space of the possible solutions can get exponentially large with the number of agents and capabilities in the system. A constraint solver is only applicable for systems with few components. Here we are looking into solutions where clustering properties can be utilized, which may lead to the possibility that local reconfiguration is possible.

Until now, we mainly considered self-reconfiguration after a failure and self-adaptation to another linear task. The next steps here are to adapt to new agents and to extend the approach to also allow for non-linear tasks (i.e. the task is no longer a totally ordered list of capabilities but rather a directed graph). An example for this type of task is a scenario where resources (for example workpieces in production or sensor data) are aggregated or split into pieces.

5 Related and Future Work

5.1 Related Work

There already exist a lot of work on constructing and specifying reconfiguration algorithms. Bussmann et al. [2] for example use agent technology for a production scenario. This differs from our approach as it is more directed to increasing throughput and not to make the system more dependable to failure of components.

In [22] a model-based development framework for self-adaptive embedded programs is presented. In contrast to our method where only the specification is necessary, the reconfiguration mechanism must directly be given. As a consequence this approach is less generic and only applicable to a limited number of domains.

The presented approach has strong relationships to the ideas and concepts of agent-based systems. Bauer et al. [?] also examined agent systems, Organic Computing systems (as well as Autonomic Computing systems), state commonalities and the divergences between them. They propose a common view on these technologies and show how they can benefit from each other with regard to software engineering (SE). They did not focus on formal aspects like specification and formal analysis of system properties.

The SCRAM method [21] is an approach for building fail safe systems based on reconfiguration. The core idea is to (ex-)change functional modules at runtime whenever necessary. While the basic mechanism is a little similar to restoring invariants, SCRAM is very much focused on safety and has only limited flexibility with regard to the class of target systems.

Kramer and Magee [4, 10] are using architectural constraints to describe the system structure, which allows for implicit management of the structure. These constraints are extracted from a system specification in Darwin ADL [11]. They focus on the system architecture and consider system structure as directed graph in which components are nodes and arcs specify bindings between a service required of a component and provided by the other. This bindings are described through the architectural constraints.

5.2 Future Work

The presented approach offers a lot of opportunities and possible next steps. One opportunity is of course to integrate specification of invariants into the software engineering process. The hope is that this can be done with OCL[14] (or with minor extensions to it). Another big chance is to use it for early evaluation of systems by building and analyzing executable models. An example of this idea for the adaptive production cell is published in [5]. Here invariants and abstract system models are used to determine the amount of failure tolerance of a self-x system in comparison to a traditional one. The obvious extension of this work would be to generalize this such that other self-x properties can also be measured.

An open question is the correctness of invariants. For a given system and some intended behavior it is in general

hard to decide if a set of invariants correctly distinguishes productive and self-x states. A good next step could be to define guidelines for specific classes of systems. The ultimate goal is then to provide a framework in which correctness of the invariants with respect to a specification of the intended behavior can be shown. Another open issue is the question of refinement. Invariants (as described in this paper) reason about an abstraction of the real implemented system (with protocols, handshakes etc.). It is not guaranteed, that properties of reconfiguration which hold for the abstract system, hold for the real system as well. The idea is here to define a generic refinement and prove a refinement relation (which at least holds for some classes of invariants).

6 Conclusion

The *Restore Invariant Approach* is of great help for the analysis, design, modeling and verification of AC/OC systems with self-x capabilities. Many self-x properties can be mapped to this approach. Self-adaptation means that the requirements change and the system tries to adapt in such a way that the new invariants hold again. Self-configuration means that if system properties change, i.e. new agents for the system, the changed system then has to fulfill the invariants again. Self-healing means that environmental assumptions no longer hold, the specified invariants are violated and the system then tries to reestablish these invariants.

The close coupling of this approach with the Organic Design Pattern allows for incorporation of self-x principles and verification early in the design phase of the system. The abstraction of the reconfiguration mechanism into the invariant to be monitored and restored allows for giving functional guarantees about a system without having a specific reconfiguration scheme. This means that every reconfiguration algorithm that is capable of restoring the invariant is correct wrt. the system specification.

Our future work will include implementing different reconfiguration strategies into realisations of the ODP. We are currently working on expressing the *Restore Invariant Approach* in the relational first-order model finder Kodkod [23]. The implementation of the organic production cell will be done in the Microsoft Robotics Framework which allows for simulation of robot kinematics and physics.

References

[1] Bernhard Bauer and Holger Kasinger. Aose and organic computing - how can they benefit from each other? In *AOIS*, pages 154–167, 2005.

[2] A. Bouajila, J. Zeppenfeld, W. Stechele, A. Herkersdorf, A. Bernauer, O. Bringmann, and W. Rosenstiel. Organic computing at the system on chip level. In *Proceedings of the IFIP International Conference on Very Large Scale Integration of System on Chip (VLSI-SoC 2006)*. Springer, October 2006.

[3] S. Bussmann and K. Schild. Self-organizing manufacturing control: An industrial application of agent technology, 2000.

[4] Edmund M. Clarke. Model checking. *Lecture Notes in Computer Science*, 1346:54–??, 1997.

[5] Ioannis Georgiadis, Jeff Magee, and Jeff 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.

[6] M. Güdemann, F. Ortmeier, and W. Reif. Safety and dependability analysis of self-adaptive systems. In *Proceedings of ISoLA 2006*. IEEE CS Press, 2006.

[7] Matthias Güdemann, Andreas Angerer, Frank Ortmeier, and Wolfgang Reif. Modeling of self-adaptive systems with SCADE. In *Proceedings of the 2007 IEEE International Symposium on Circuits and Systems, ISCAS07*. IEEE, 2007.

[8] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

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

[10] J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Computer*, January 2003.

[11] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[12] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

[13] C. Müller-Schloer. Organic computing: on the feasibility of controlled emergence. In *CODES+ISSS '04*, NY, USA, 2004. ACM Press.

[14] Florian Nafz, Frank Ortmeier, and Hella Seebach. Video of the MSRS simulation, 2007. http://loewenheim.informatik.uni-augsburg.de/kiv/msrs.avi.

[15] OMG Object Management Group. UML 2.0 OCL Specification, 2003.

[16] A. Pnueli. System specification and refinement in temporal logic. In R. Shyamasundar, editor, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 652, pages 1–38. Springer-Verlag, Berlin, Germany, 1992.

[17] Holger Prothmann, Fabian Rochner, Sven Tomforde, Jürgen Branke, Christian Müller-Schloer, and Hartmut Schmeck. Organic control of traffic lights. In Chunming Rong, Martin Gilje Jaatun, Frode Eika Sandnes, Laurence T. Yang, and Jianhua Ma, editors, *Proceedings of the 5th International Conference on Autonomic and Trusted Computing (ATC-08)*, volume 5060 of *LNCS*, pages 219–233. Springer, 2008.

[18] Frank Reichenbach, Andreas Bobek, Philipp Hagen, and Dirk Timmermann. Increasing lifetime of wireless sensor networks with energy-aware role-changing. In *Proceedings of the 2nd IEEE International Workshop on Self-Managed Networks, Systems & Services (SelfMan 2006)*, pages 157–170, Dublin, Ireland, 2006.

[19] Urban Richter, Moez Mnif, Jürgen Branke, Christian Müller-Schloer, and Hartmut Schmeck. Towards a generic observer/controller architecture for organic computing. In Christian Hochberger and Rüdiger Liskowsky, editors, *INFORMATIK 2006 – Informatik für Menschen*, GI-Edition – Lecture Notes in Informatics, pages 112–119, Bonn, Germany, September 2006. Köllen Verlag.

[20] Fabian Rochner, Holger Prothmann, Jürgen Branke, Christian Müller-Schloer, and Hartmut Schmeck. An organic architecture for traffic light controllers. In Christian Hochberger and Rüdiger Liskowsky, editors, *INFORMATIK 2006 – Informatik für Menschen*, GI-Edition – Lecture Notes in Informatics, pages 120–127, Bonn, Germany, September 2006. Köllen Verlag.

[21] Hella Seebach, Frank Ortmeier, and Wolfgang Reif. Design and Construction of Organic Computing Systems. In *Proceedings of the IEEE Congress on Evolutionary Computation 2007*. IEEE Computer Society Press, 2007.

[22] Elisabeth A. Strunk and John C. Knight. Dependability through assured reconfiguration in embedded system software. *IEEE Transactions on Dependable and Secure Computing*, 3(3):172–187, 2006.

[23] Li Tan. Model-based self-adaptive embedded programs with temporal logic specifications. In *QSIC '06: Proceedings of the Sixth International Conference on Quality Software*, pages 151–158, Washington, DC, USA, 2006. IEEE Computer Society.

[24] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. pages 632–647. 2007.