Design and Construction of Organic Computing Systems

Hella Seebach, Frank Ortmeier and Wolfgang Reif
Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, Universitätsstrasse 14, D-86135 Augsburg
{seebach, ortmeier, reif}@informatik.uni-augsburg.de

Abstract—The next generation of embedded computing systems will have to meet new challenges. The systems are expected to act mainly autonomously, to dynamically adapt to changing environments and to interact with one another if necessary. Such systems are called organic. Organic Computing systems are similar to Autonomic Computing systems. In addition Organic Computing systems often behave life-like and are inspired by nature/biological phenomena.

Design and construction of such systems brings new challenges for the software engineering process. In this paper we present a framework for design, construction and analysis of Organic Computing systems. It can facilitate design and construction as well as it can be used to (semi-)formally define organic properties like self-configuration or self-adaptation. We illustrate the framework on a real-world case study from production automation.

I. INTRODUCTION

A new trend in computer science is to make systems organic. Organic here means, that the systems are capable of autonomously reacting to changes in their environment. Such capabilities are called self-organising, self-healing, self-configuring, self-adapting or simply self-x. The basis for a self-x capability is often to give the systems some degrees of freedom which allow them to react to component failures and/or changing environments. To implement organic behavior all system components must be enhanced such that they can communicate their functional capabilities and reflect the capabilities of other components.

In Sect. II an organic design pattern is given which allows for systematic design of a broad class of Organic Computing systems. Sect. III shows possible applications in the development process. An exemplary application to a real-world case study is presented in Sect. IV. Sect. V discusses future work and Sect. VI concludes the paper.

II. FRAMEWORK

Many Organic Computing systems can be divided into three structural parts: productive part, self-x infrastructure and organic control mechanism. Therefore, design challenges extend into three dimensions. One dimension subsumes domain specific, functional parts (in general the original productive system), the second dimension contains organic and self-x infrastructure of the system (like communication channels etc.) and the third dimension is an observer/controller layer which, if necessary (e.g. when component failures occur), coordinates the system components and their self-x properties. System design and development will become a lot easier if these dimensions can be split up.

The presented framework for designing and modelling Organic Computing systems contains three different facets. An architectural view (see Sect.II-A) supports adding self-x infrastructure to system components. Component interaction and system structure is modelled as application of an organic design pattern (see Sect.II-B). Organic control and reconfiguration mechanisms are encapsulated in a specific design block – the virtual centralised observer/controller (see Sect.II-C).

A. Architecture

Fig. 1 shows the architecture of Organic Computing systems. The productive system allocates the basic functionality, especially the components, which care for the functional behaviour of the organic systems. The self-x infrastructure provides an infrastructure which is needed to enhance a normal productive system to an organic system. The self-x infrastructure is responsible for the following three main tasks: First of all it wraps components of the productive system into agents which allows them to communicate with each other. Secondly it allows agents to announce their capabilities, so that every agent has information about other agents with which it can interact and which tasks they could perform. The agents must be able to reconfigure their usage of capabilities according to the roles they have to perform. Therefore the third task of the self-x infrastructure is to provide interfaces for the agents to enable them for reconfiguration. The self-x infrastructure is the basis for the observer and controller.

Fig. 1. Architecture for Organic Computing systems

The observer/controller layer is responsible for managing and (re-) configuring the components of the system in an intelligent way. This is typically done by continuously observing the system and taking control actions whenever necessary. Therefore this layer contains an organic control mechanism and is called observer/controller (O/C). In many Organic Computing applications, a centralised O/C layer is not possible or undesirable. The implemented or running
Different approaches for decentralising observer/controllers on the agents are conceivable, namely classical leader election algorithms, global state propagation [1] [2], planning algorithms based on the formal model of the system [3][4] or by distributed constraint-solving [5][6].

Fig. 2 shows the decentralised architecture of an organic system. The arrows represent the interaction between O/C layers of the system’s components. The realization of the interaction between the components (discovery, etc.) is described in the next section.

There exist already different approaches for Observer/Controller architectures and control algorithms in the fields of Organic Computing and Autonomic Computing. Most of these can be integrated in the observer/controller layer of the presented architecture. For example in [7] an observer/controller architecture is presented and in [8] it is used for organic traffic light controllers. This architecture can be realized in a central, decentralised or a multi-level way as depicted in [9].

Another approach - the vision of Autonomic Computing - is depicted in [10] and the importance of the monitoring and control of such systems is described. Autonomic elements continually monitor their own state and their suppliers by an autonomic manager. The autonomic elements consist of at least one managed element (storage, CPU, service,...) and the manager. Based on knowledge, monitoring data and analyzed effects, the autonomic manager decides how the element has to behave and observes the execution of the corresponding plan. If one element detects a fault or inconsistency it tries to solve the problem with the help from the other autonomic elements. All these approaches seem promising to be integrated in the presented architecture.

B. Organic Design Pattern

Design and construction of Organic Computing systems is often challenging. To facilitate the modelling of these systems, an organic design pattern (ODP) has been developed. This pattern suits for characterization and design of Organic Computing applications. It gives explicit descriptions (constructs and rules) of how domain-specific models generally look like and how they should be built. Specific applications are then instances of the ODP. The pattern instructs both the structure of the whole system and the communication channels of system components.

Fig. 3 shows the ODP. The main components of Organic Computing systems are Agents. They process Resources according to given Tasks. Every Agent knows a set of Agents from which it can receive Resources (Input), a set of Agents to whom it can give Resources (Output) and a set of Capabilities which it can provide. Which Capabilities an Agent performs and from where resp. where to Resources are taken resp. given is captured in its Role.

Inter-agent communication is done by Messages. It is convenient to distinguish between AgentMessages and ExternalMessages. The first are generated on purpose by some agent while the other describe (uncontrollable) inputs from the environment. Note that the component agent represents both the hardware and the software. One part of the hardware of a robot is for example a collection of sensors, which communicates with the software to announce a lost capability (e.g. “broken drill”). This announcement is one example for an ExternalMessage.

To ensure consistency the Role may be restricted by OCL-constraints [11] which assert for example (constraint 1) that the role assigned to an agent only includes capabilities the particular agent can perform. The constraints 2 and 3 assert that an agent can only receive resources from and give ressources to agents corresponding to its input and output associations. The associations input and output are also restricted by OCL-constraints. The constraints 4 and 5 assert that if agent A has agent B in its input-association, agent B must have agent A in its output-association.

In the adaptive production cell case study (see Sec. IV-A), concrete instances of the Agents are robots and autonomous carts. Resources are the workpieces which are to be processed with the Task “drill then insert then tighten”. Roles of the agents are for example: “take from cart A - drill - give to cart B” or “take from robot A - transport - give to robot B”. It is useful to also refine the OCL constraints for this instantiation of the ODP.

Agents, Capabilities, Resources and Tasks are all part of the functional aspect of the system and the self-x infrastructure. They describe what has to be done and what can be done. But the whole system can only process Resources according to their Tasks if Roles are distributed correctly among Agents. Distribution of Roles to Agents is the core of the organic part of the system. For finding a correct distribution it is necessary to take Capabilities, Tasks and agent topology (i.e. the connection between agents through their inputs and outputs) into account. This is a job of the reconfiguration algorithm which is part of the virtual centralised Observer/Controller.

C. Virtual centralised Observer/Controller

After designing the functional parts of the system and adding self-x infrastructure, the reconfiguration algorithm has to be defined. The following questions must be answered: When and why should the system start a reconfiguration? In
which state must the system be after reconfiguration? Which constraints must always hold?

These question are typically answered by (i) defining observer components, which continuously monitor the system, (ii) adding control components which use an intelligent (possibly distributed) reconfiguration algorithm, that takes the state of the system as input and calculates a new working configuration and (iii) by analysing the system together with its observers and controllers. This approach often involves a lot of work and only allows for analysing the system after observers, controllers and reconfiguration strategies have been implemented.

The presented framework offers the possibility of a different approach. Necessity of reconfiguration can be formally expressed as the violation of a logical formula – which contains the actual configuration and capabilities of the agents as well as intended tasks as free variables. In the framework reconfiguration means finding a correct allocation of roles to agents such that this formula will hold again and spreading this new distribution to the agents. So the observing part of the OC system can be modelled as evaluating a formula, the reconfiguration strategy can be modelled as a function whose results fulfil certain constraints and the controlling part is a set of state changes.

To capture this, the concepts VirtualCentralisedObserver-Controller (O/C) and RoleAllocation are introduced. The first one represents a specification (e.g. a set of temporal logic formulas) of an algorithm to solve the role allocation problem. The output of this component is a RoleAllocation which defines Agents and their Roles which they should perform.

This structure already makes formal analysis of the system possible. A results is, that the specification of the reconfiguration algorithm (together with the functional model of the system – i.e. Agents, Capabilities, Tasks, etc.) can be used to analyse the system and give guarantees on its behaviour. So this approach allows for analysing the quality of the system with using only the specification of a (decentralised) observer/controller architecture rather than its implementation. This is beneficial in three ways. It allows to do top-down design of the system and evaluate functional at very early design stages. Secondly it separates functional design and design of OC features. Finally existing OC algorithms for self-configuration, role switching, etc. may be plugged in as long as these algorithms fulfil the specification which is captured in the VirtualCentralisedObserverController.

III. USING THE FRAMEWORK

The organic design pattern presented in Sect. II is a formal foundation for numerous aspects related to the development of Organic Computing systems. It helps formally defining self-x properties, it gives a guideline for extending standard systems with self-x capabilities, it supports easy implementation of Organic Computing systems and it gives
a formal basis for verification. In this section several Organic Computing aspects are shown and possible connections to the framework are sketched.

A. Formalizing self-x properties

An often discussed question in Organic Computing is, how to precisely define self-x capabilities. When may a system be called self-adaptive or self-healing? In this context the organic design pattern can help, by grounding the self-x concepts on the underlying entities of the system. Goals are properties of the system which should be fulfilled. One example is that resources are treated according to their tasks, another example is safety. This leads to the following definitions:

Definition 1: A system SYS, which is modelled as an instance of the organic design pattern is called

- **self-configuring** for a goal G, if the system is put into running mode with an arbitrary role allocation \( \sigma_{arb} \), then it will eventually come to a role allocation \( \sigma_G \) in which G will be achieved.
- **self-adapting** for a given set \( T = \{ t_i \} \) of tasks, if there is a change of tasks from \( t_1 \) to \( t_2 \) and \( t_1, t_2 \in T \), then the system will eventually come to a role allocation in which the new task \( t_2 \) will be achieved
- **self-healing** for a given set \( C \) of capabilities and a goal \( G \), if after failure/loss of any capability \( c \in C \), then it will eventually come to a role allocation in which \( G \) will be achieved again (as long as this is theoretically possible).
- **self-optimizing** for a given goal \( G \) and a given rating function \( f : \Sigma \rightarrow \mathbb{R} \) (where \( \Sigma \) denotes the space of all eligible role allocations), if the system eventually comes to a role allocation \( \sigma \) in which \( f(\sigma) \) is (locally) minimal over the set \( \Sigma \).

These definitions are very handy for reasoning about Organic Computing systems, which are built as application of the presented pattern. Self-configuration is then simply the question: if the system’s initial role allocation is random, will the system then be able to trigger a reconfiguration and find a correct role allocation to achieve the goal? An example for self-adaptation is, if tasks change within a certain bandwidth (maybe defined by a predicate \( P \)), the system reconfigures and finds a correct role allocation for the new task.

B. Making systems OC-ready

Another topic is to define a process for systematically enhancing traditional systems with self-x capabilities. The goal is to standardise the development process. An important property of the process is that it allows splitting between design and construction of productive parts and organic parts. This is of importance because in this scenario a productive system already exists and should be only enhanced by adding organic behavior. A possible guideline could be of this form:

1) Start with a traditional productive system and analyze this system to identify its components and the collaboration structure of these components.

2) Identify groups of similar components. These groups will later become subclasses of the abstract class Agent of the organic design pattern.

3) Enhance all classes of components with the self-x infrastructure as described in Sect. II-A. This makes them effectively to agents which can announce their capabilities and communicate with each other.

4) Individual components of the system become instances thereof.

5) Identify analogously the resources and tasks in the traditional productive system.

6) Add an O/C layer to the whole system which monitors and (whenever necessary) reconfigures the agents.

Note that the quality of the system and the degree of “organic behavior” in the system is determined by the quality of the O/C layer. A trivial O/C layer is that there is no monitoring at all and only one fixed configuration is set (e.g. the configuration which describes the interaction structure of the initial productive system). This is basically the original productive system. An intelligent one will use smart reconfiguration algorithms and perform better.

IV. Case Study

This section describes an application from production automation. The main concepts of the modelling framework and analysis methods of Sect. II and Sect. III will be explained in this example. In this section only a brief and informal description is given. A detailed report on the application and safety/self-healing related questions may be found in [12][13].

![Fig. 4. Standard role allocation of the production cell](image)
goal of the cell is to process workpieces following a given specification. Every robot can accomplish three tasks: drilling a hole in a workpiece, inserting a screw into a drilled hole and tightening an inserted screw. These tasks are done with three different tools that can be switched. One scenario is, that every workpiece must be processed by use of all three tools in a given order (1st: drill, 2nd: insert, 3rd: tighten = DIT). Workpieces are transported from and to the robots by autonomous carts. Changing the tool of a robot requires a lot of time (compared to performing the task itself). Therefore the standard role allocation of the system is to spread out the three tasks between the three robots, and the carts transfer workpieces accordingly. This situation is shown in Fig. 4.

B. The application as instance of the design pattern

In Fig. 5 an instantiation of the design pattern for the organic production cell is shown. Some classes in this figure carry a link to the corresponding super element of the design pattern in the upper right corner. The production cell comprises three types of Agents: Robots, AutonomousCarts and Storages. Each agent encapsulates the functionality of the corresponding functional part of the system and has been enhanced by a self-x infrastructure as described in Sect. II-A. This means they gain the ability to communicate with their environment and other agents - in this case other robots, carts or storages.

Furthermore they can, if necessary, reconfigure and perform new roles according to their capabilities. The capabilities of a robot are Tools, an autonomous cart can Transport workpieces and a storage can Store workpieces. Each robot is equipped with a Drill, an Inserter and a Screwdriver. Therefore Robot agents get corresponding capabilities: Drill, Inserter and Screwdriver. Due to the nature of the system Workpieces (instances of Resource) can only be given from Robots (or Storages 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 with the Resources. In the example: “first drill a hole (use Drill), then insert a screw (use Inserter) and finally tighten the screw (use Screwdriver”). A RobotRole defines which tool a robot has to use, from which carts it is supposed to pickup workpieces and to which carts it should give the workpieces.

This case study showed, that the organic design pattern can be applied to organic applications without difficulty. It turned out that the separation of functional and organic aspects during the design process is possible and very useful. Note that only the static aspects of the system are modelled and the reconfiguration is captured in the VirtualCentralizedObserverController. The next step to be done is to extend the pattern, especially the VirtualCentralizedObserverController, with adequate software engineering methods to include the currently missing dynamics in the development process.

C. Formalizing self-x properties

Although small, the example exhibits several aspects of self-x properties and especially self-configuration which we regard as building block for most, if not all self-x properties.

a) Self-Configuration: Using the definition of Sect. III the system may be called self-configuring, if the system
can automatically find a role allocation for a given goal G. This question is directly connected to the correctness of the reconfiguration algorithm (which is modelled in the virtual centralized observer/controller). In the example this means, that the roles are allocated such that (1) at least one robot is configured to use its drill, inserter and screwdriver and that (2) the carts are configured correctly (with respect to the role allocation of the robots) to transport the goods correctly.

b) Self-Adaptation: Self-adaptation takes place, when for example partly processed workpieces enter the production cell. Assume that a new group of workpieces must be processed which only need a hole drilled and nothing else. The system can be called self-adapting to this task (or more precisely to the set of the initial task and this new task), if the reconfiguration algorithm now finds a new role allocation, such that the roles are distributed accordingly to this new task. Here this new task would only be D (drill).

c) Self-Healing: An interesting new situation occurs when one or more tools break and the current role allocation allows no longer correct DIT processing of incoming workpieces. In Fig. 6 the drill of the left robot (1) broke and DIT processing is not possible, as no other robot is configured to drill. A traditional production cell would probably come to a halt and wait for maintenance. It is obvious that this is not a good reaction, as the robots have three tools each and can switch to another tool if one breaks. So it should be possible for a self-healing system to detect this situation and reconfigure itself in such a way, that DIT processing is possible again. This implies that at least one other robot also has to switch its tool, so that all three processing steps are possible again (see Fig. 7).

Another form of self-healing that can be illustrated within this case study is graceful degradation. This means that the system tries to fulfill at least parts of its functions as long as possible. This could be drilling holes in a workpiece and inserting screws, but not tightening them if all screwdrivers are broken. Another form would be to use one robot to accomplish more than one task. This also falls under graceful degradation because it preserves the functionality but reduces the throughput of the production cell considerably (under the assumption that switching tools takes quite a while).

d) Self-Optimization: A correct role allocation (σ₁) is to let every robot process the workpieces completely. Is this role allocation better then the role allocation were the tasks are split? Assume the system must optimize its performance. Performance is measured by a real function f which measures the time a workpiece needs to be processed. In the above role allocation the time to transport (t_{trans}) the workpieces does not matter because the workpieces are transported in large pallets, which makes this time negligible. The time needed to process one workpiece is calculated as the time a single robot needs for doing all the work (including the time t_C needed to switch tools between different tasks). So the whole cell with three robots in average needs \( f(\sigma_1) = (t_C + t_D + t_I + t_I + t_C + t_T)/3 \) seconds to process one piece. If the cell produces in a role allocation \( \sigma_2 \), which spreads the tasks between all three robots then \( f \) will be given as the maximum of all times needed in the production chain. Possible values for the example are: \( t_D = 4, t_I = 2, t_T = 5, t_C = 10 \). So the role allocation where one robot does all the work will need \( f(\sigma_1) = 13.6 \) seconds for one piece while the other role allocation \( \sigma_2 \) will need \( f(\sigma_2) = 5 \) seconds per piece (which is also optimal for this scenario). The system is called self-optimizing for a given function f, if the reconfiguration algorithm calculates role allocations such that \( f \) is minimal.

So the reconfiguration algorithm has to find a valid role allocation as before. The difference to self-adaptation is that the “has”-relationship between robots/carts and their capabilities has changed. The algorithm must take this into account. The system may be called self-healing if still valid role allocations are found.

---

1 Here, \( t_D, t_I \) and \( t_T \) denote the time needed to drill, insert and tighten.
V. OUTLOOK

Most OC systems only differ in their functionality, used hardware and their reconfiguration algorithms. They all share the necessity that individual components (agents in the framework!) find each other and communicate with each other. We think, this interaction infrastructure between the agents (agent discovery, messaging, etc.) can possibly be handled by existing multi-agent frameworks (e.g. AgentService, SimAgent [14], JACK [15]) or convenient Publish/Subscribe systems (e.g. [16], [17], [18]). With the presented framework as starting point we began an implementation and verification of the adaptive production cell. We are currently working on building a simulation of the adaptive production cell. The hardware will be simulated within Microsoft Robotics Studio [19]. It provides a virtual, service-oriented simulation environment for robotic applications. The controlling software will be implemented as described in Sect. IV-B. As Microsoft Robotics Studio is based on the .NET framework we decided to use the multi-agent framework AgentService [20] for the inter-agent communication which supports agent development based on the programming language C#. It seems to facilitate wrapping components into self-x infrastructure a lot.

Another topic we work on is to use the ODP as formal basis for verification. Instantiations of the ODP may be used to verify properties of the system before actual implementation. This will become possible if abstraction of the hardware, contained in the Agents, is integrated into the model. A specification of the role allocation algorithm can be encapsulated in the VirtualCentralisedObserver/Controller. Verification of this model is “easier” (compared to a complete model of the system) because firstly the level of abstraction is very high and secondly a central specification of a decentralised reconfiguration algorithm helps a lot in formal reasoning environments. It is clear that correctness of the so found results only holds for the implemented system if the implementation fulfills some refinement relationship.

VI. CONCLUSION

We presented a generic framework, which allows for modelling a broad class of Organic Computing systems. This framework can be a great help for defining self-x properties as well as for implementing a system such that it shows organic capabilities. The core idea is to separate development of functional and organic properties. This also allows for systematically wrapping existing systems in an organic infrastructure and stepwise enhancing their capabilities for reorganization. This modelling paradigm also helps for rating the possible benefit of an organic architecture at very early design stages. It allows for using the specification of a reconfiguration algorithm for verification and thus for calculating potential benefits.

We illustrated the results on a real-world case study from production automation and presented results for this case study.

REFERENCES