IMPLEMENTING ORGANIC COMPUTING SYSTEMS WITH AGENTSERVICE*

Florian Nafz, Frank Ortmeier, Hella Seebach, Jan-Philipp Steghöfer and Wolfgang Reif
Lehrstuhl für Softwaretechnik und Programmiersprachen, Universität Augsburg, D-86135 Augsburg, Germany

Keywords: Organic computing, software engineering, design methods, multi-agent systems.

Abstract: Designing and implementing Organic Computing systems is typically a difficult task. To facilitate the construction a design pattern for Organic Computing systems has been developed. This organic design pattern (ODP) helps in modeling and designing a broad class of Organic Computing systems.
This paper focuses on the implementation of Organic Computing systems with the help of this pattern. The core idea is to provide a generic implementation by mapping ODP artifacts to artifacts of a multi-agent framework. The used framework – AgentService – is one of the few C# multi-agent frameworks. In this paper a possible implementation as well as benefits and limitations are described.

1 INTRODUCTION

A new trend in computer science is to make systems organic (Müller-Schloer et al., 2004; Branke et al., 2006). Organic here means, that the systems are capable of autonomously reacting to changes in their environment. Such capabilities are called self-organizing, self-healing, self-configuring, self-adapting or simply self-x (Güdemann et al., 2006a). The basis for 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. In (Seebach et al., 2007) an organic design pattern for modeling Organic Computing systems has been introduced. It helps a lot for modeling and design, but unfortunately artifacts of the pattern can not be directly mapped into code. This is because ODP artifacts are defined on a very abstract level. Interaction on ODP level typically comprises – on the code level – sending a message, relaying it, receiving it, interpreting it and acting according to it. To facilitate construction of Organic Computing systems it is useful to provide generic reference implementations of ODP models in a standard programming language. In this paper C# has been chosen as programming language and the multi-agent framework AgentService (Boccalatte et al., 2006) is used as communication infrastructure. Possibilities, risks and experiences of this generic implementation will be discussed in the following sections. They are illustrated on a real-world case study from the domain of production automation.

In Sect. 2 a brief introduction to the organic design pattern is given and Sect. 3 provides a similar introduction to the multi-agent framework AgentService. A generic mapping of ODP artifacts into AgentService artifacts is shown in Sect. 4. The practicability and lessons learned are illustrated on a case study from production automation (Sect. 5). Sect. 6 concludes the paper.

2 DESIGNING OC SYSTEMS

Design and construction of Organic Computing systems is often challenging. To facilitate the modeling of these systems, an organic design pattern (ODP) has been developed. This pattern is suitable 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. 1 shows the ODP. The main components of

*This research is partly sponsored by the priority program "Organic Computing" (SPP OC 1183) of the German research foundation (DFG)

Nafz F., Ortmeier F., Seebach H., Steghöfer J. and Reif W. (2008).
IMPLEMENTING ORGANIC COMPUTING SYSTEMS WITH AGENTSERVICE.
In *Proceedings of the Third International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 64-71
DOI: 10.5220/0001762700640071

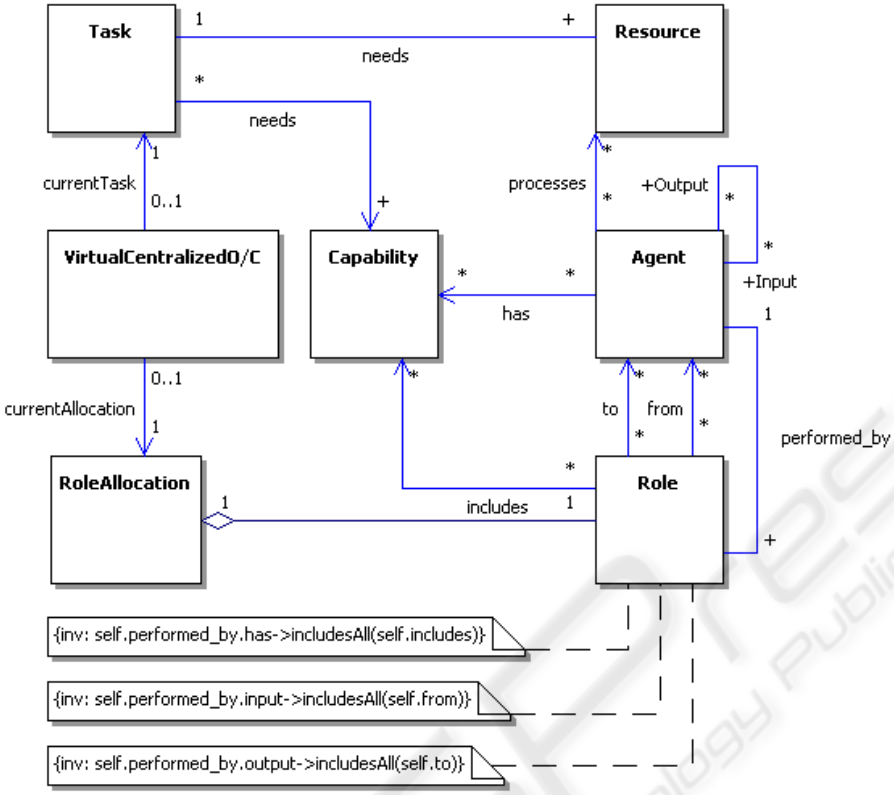

Figure 1: Organic design pattern (ODP).

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 resp. to which Agent *Resources* are taken resp. given is captured in its *Role*.

To ensure consistency *Roles* may be restricted by OCL-constraints (Object Management Group, 2003) which assert for example (constraint 1) that the role assigned to an agent only includes capabilities the particular agent can perform.

Agents, *Capabilities*, *Resources* and *Tasks* are all part of the functional aspects 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*. Allocation of *Roles* to *Agents* is the core of the organic part of the system. For finding a correct allocation it is necessary to take *Capabilities*, *Tasks* and agent topology (i.e. the connection between agents through their inputs and outputs) into account. All organic intelligence may be specified in one entity: the virtual centralized Observer/Control. This entity can then either contain a (distributed) algorithm, which calculates role allocations or just a specification of such an algorithm.

This pattern allows firstly for splitting construction of functional and organic parts of the system in two. Secondly, it allows for a generic specification of role allocation algorithms as logical problems. The basic idea here is to describe valid (i.e. working) role allocations with a logical formula, which has the degrees of freedom as free variables. Finding a valid role allocation is then equivalent to finding a satisfying valuation of free variables of a formula. See (Seebach et al., 2007) for more details. An example system which is modeled as an instance of the ODP is described in Sect. 5.

In the context of this paper it is now more important to note, that ODP provides a generic modeling formalism for Organic Computing systems. It consists of a certain set of high-level artifacts, some interaction between them and some consistency rules. The goal – in this paper – is to provide a generic transformation of an ODP model into executable code.

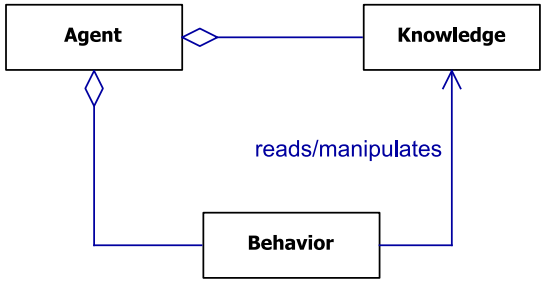

Figure 2: Basic AgentService concepts.

3 AN INTRODUCTION TO AGENTSERVICE

In this section, a short summary of the basic concepts of AgentService is given. For a more detailed overview see (Boccalatte et al., 2004). The AgentService framework has been developed at the University of Genua and is one of the few C# agent platforms. It incorporates all the basic concepts of a FIPA-compatible[2] application, i.e. an Agent Management System as well as FIPA-compliant messaging services and directory services. Furthermore, a comprehensive code basis for agent development is available in the form of the AgentService API. Additional capabilities include persistence of agent state and a plug-in system, which allows for the development of enhancements to the platform itself. A simplified overview of the most important parts of an AgentService system is shown in Fig. 2.

The central components of AgentService are *Agents*. An *Agent* comprises several instances of two different classes: *Behaviors* and *Knowledges*. *Behaviors* encapsulate all actions an agent is able to perform, including communication with other agents and manipulation of the environment and its knowledge base. *Knowledges* contain everything the agent knows about itself and its environment. In general, several behaviors per agent may be active concurrently. If – for example – an agent provides translation between file formats, then this will be encapuslated in a behavior. Several instances of this behavior may be created, such that the agent can accept multiple translation requests at the same time. These behaviors will then run concurrently and terminate as soon as they finished their individual translation job. Note that all behaviors access a common knowledge base (in this example maybe a dictionary), which consists of all knowledges known to the agent.

Before agents can exchange messages, a conversation between the two peers has to be established. A conversation provides a context for the message exchange and enables agents to subsume different messages to one instance of the protocol sequence. Incoming conversations are accepted by the agent and then forwarded to one of its behaviors which in turn handles the protocol. Messages can be transferred within these conversations by means of a proprietary message format based on C#'s serialization facilities or as standardized FIPA-ACL (FIPA ACL, 2002) message codes.

Every time a behavior wants to access a knowledge it has to lock the object first. This enables different behaviors of an agent to concurrently manipulate the shared understanding of their environment. Knowledges may contain arbitrary classes and are accessed with methods provided by the agent framework. Knowledge objects can automatically be persisted by the framework to provide recovery in case of a system crash. Configuration of the platform and the agents is contained in XML-files. These files describe the basic parameters of the framework as well as the correct instantiation of knowledges and behaviors.

4 IMPLEMENTING OC SYSTEMS WITH AGENTSERVICE

This section details a generic transformation of ODP models into code using the AgentService framework. Therefor, abstract ODP artifacts must be mapped in a generic way to artifacts of AgentService framework. An example of a concrete mapping (as an instance of the generic mapping) will be presented in Sect. 5.3. Table 1 gives an overview of the mapping:

There is a direct relationship between ODP's notion of an *Agent* and the agent artifact in AgentService. Both entities are used for message dispatching and as containers for the core functionality. An important difference to ODP is that AgentService does not explicitly state which other agents are inputs or outputs for resources. These associations are implicitly contained in the configuration of the instance. ODP allows constraints on possible work-flows by limiting the associated agents. In the implementation, each agent is allowed to exchange resources with each other hypothetically.

[2]The Foundation for Intelligent Physical Agents (FIPA) defined the industry standard for interoperable agents. With the completion of the standard in 2001 a common basis for communicating, mobile agent systems has been set to which almost all modern agent platforms comply. The standard includes definitions for protocols, message formats as well as architectural foundations for agents systems. Details can be found on the FIPA website (FIPA Website, 1996).

Table 1: Relation between ODP and AgentService artifacts.

ODP Artifact	**AgentService Artifact**
Agent	Agent
VirtualCentralizedO/C	Agent
Capability	One generic behavior, which contains all possible capabilities
Role	Knowledge, which parameterizes the generic behavior
Role Allocation	Set of knowledge
Task	Need not be mapped explicitly, for self-healing and self-configuring.
Resource	Knowledge

The *Capabilities* of the agent are contained in one generic behavior: the DO behavior. It contains all the logic for processing resources. It is generic in the sense, that it contains all theoretically possible capabilities this type of agent may have. The DO behavior is typically restricted with a "has" knowledge. The "has" knowledge corresponds to the *Has* association of the ODP. Typically, an agent initially has all capabilities its class provides. During runtime, certain capabilities might vanish, because an agent is no longer able to perform these certain tasks. In such a case, the HAS knowledge is manipulated to reflect the loss of the capability and the application usually has to be reconfigured to become functional again.

Note, that in addition to ODP in AgentService some more communication infrastructure is needed for passing *Resources* between *Agents*. While on ODP-level this can simply be done by changing associations, on code-level it is necessary to implement some handshake protocols. Therefore, agents incorporate two additional basic behaviors (besides the DO behavior): GIVE and TAKE. GIVE and TAKE are used to exchange resources between agents consistently. They implement an exchange protocol which basically consists of a handshake, the transmission or reception of the resource itself and a concluding acknowledgment. The GIVE and TAKE behavior can be restricted with knowledges (i.e. to whom resp. from whom resources may be given/taken). This models the *Input* and *Output* associations of the ODP.

A *Role* is also implemented as a knowledge. This knowledge describes which *Capability* of the generic DO behavior the agent is currently configured to perform. It also states from which agents resources should be accepted and to which they are to be given. This is done by setting specific knowledges, which configure (together with the knowledges representing the *Input* and *Output* associations) the GIVE and TAKE behaviors. *Role allocations* are in consequence a set of (role) knowledges. Summarizing a role in AgentService means: (1) accepting certain resources (with the TAKE behavior), (2) processing them (with the DO behavior) and (3) relaying them (with the GIVE behavior)[3].

Tasks are not explicitly translated. This is because the current implementation aims only at self-healing and self-configuring systems. Therefore tasks do not change dynamically. In the current implementation they are explicitly part of the virtual Observer/Controller agent. In future additions it seems to be possible to map them to specific knowledges.

The *Resources* which are processed by the agents are mapped to knowledges as well. Resources are exchanged between agents according to the roles. Technically this is done using the GIVE and TAKE behaviors.

5 CASE STUDY

This section illustrates the presented transformations on a real-world case study. The case study is an application from production automation. In this paper only a brief and informal description of it is given. A more detailed description of the application and a report on safety/self-healing related questions may be found in (Güdemann et al., 2006b; Güdemann et al., 2006).

5.1 Application

The example describes a vision of tomorrow's production systems. In contrast to a traditional production cell, where the interaction between robots is fixed, a new adaptive production cell will dynamically change its interaction schemes. In the example, assume a production cell consisting of three robots, which are connected with autonomous transportation units.

[3]Not all agents have all three parts. For examples agents which produce or consume resources do not have TAKE resp. GIVE.

The functional 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 with all three tools in a given order (1st: *Drill*, 2nd: *Inserter*, 3rd: *Screwdriver*). Workpieces are transported from and to the robots by autonomous carts. Changing the tool of a robot requires a lot of time (compared to using it). 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. 3.

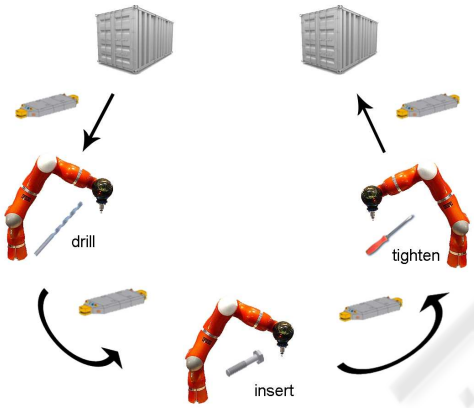

Figure 3: The adaptive production cell.

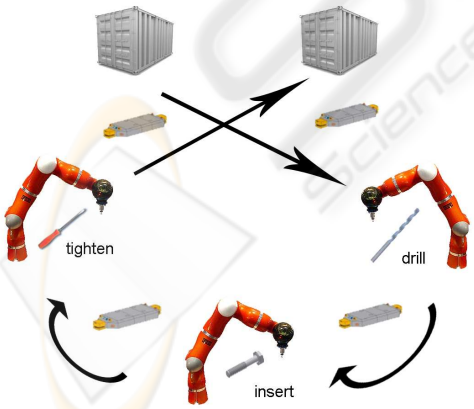

Figure 4: Reconfigured production cell.

If now one robot has some kind of defect (for example the drill of robot 1 brakes), then the cell can reconfigure itself, such that production is again possible. This situation is shown in Fig. 4. Robot 3 and Robot 1 have switched their jobs and the carts changed their routes accordingly. Other reconfigurations work analogously.

5.2 The Application as Instance of the Design Pattern

This example system is now designed using the ODP. Fig. 5 shows the resulting design as an instantiation of the organic design pattern. 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 model comprises three types of *Agents*: *Robots*, *AutonomousCarts* and *Storages*. Each agent encapsulates the functionality of the corresponding functional part of the system.

The capabilities of robots are *Tools*, autonomous carts can *Transport* workpieces and storages 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 modeled and the reconfiguration is captured in the *VirtualCentralizedO/C*. The next step to be done is to extend the pattern, especially the *VirtualCentralizedO/C*, with adequate software engineering methods to include the currently missing dynamics in the development process. A more detailed discussion on modelling this case study as an instance of ODP may be found in (Seebach et al., 2007).

5.3 The Case Study in AgentService

The generic mapping of ODP to AgentService was applied to this case study. Exemplary for all parts of the case study, the mapping of robot agents is described in a little more detail now. As mentioned above, robot agents have three different capabilities (*Drill*, *Inserter*, *Screwdriver*). They can give and take

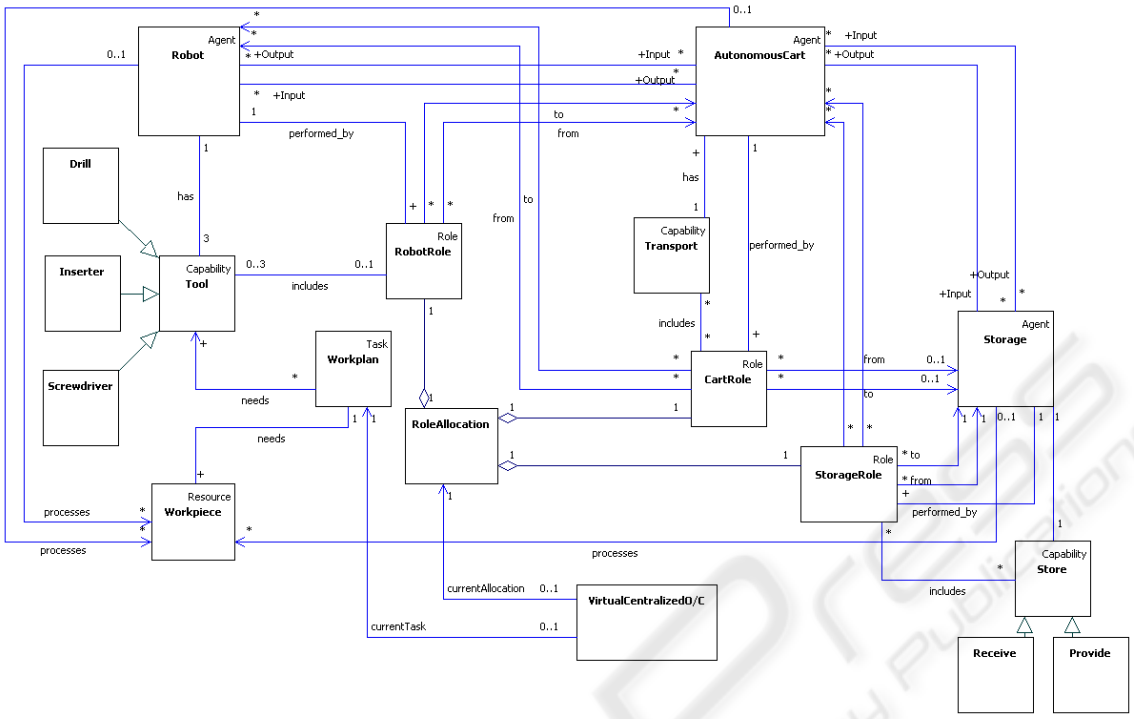

Figure 5: Instantiation of the organic design pattern for the adaptive production cell.

instances only from and to carts and they will perform their assigned role, whenever a workpiece is available. If a tool brakes the robot triggers a reconfiguration of the production cell which ultimately leads to a new role allocation by the virtual Observer/Controller. The following paragraphs show how this behavior is modeled in AgentService with the help of ODP artifacts.

Every agent contains a message filter which dispatches incoming conversations to the behavior which handles the protocol case. This filter receives the first message of a conversation and creates – depending on the message's content – a new behavior instance which will handle the protocol. E.g. a *Cart* connects to a robot and sends a "READY-FOR-UNLOADING" message, thus indicating that it is currently carrying a workpiece. The robot acknowledges the message and starts the *TakeWorkpieceFromCart* behavior which is an instance of the abstract *TAKE* behavior outlined in section 4. The agent waits until a currently processed workpiece has been retrieved by another cart and then initiates a new conversation to the waiting cart. It signals that it is ready to get the workpiece from the cart, the workpiece is transmitted and the cart is dismissed. After the workpiece has been received, the behavior *ProcessWorkpiece* (*DO*) is started. Now, the workpiece is transformed according to the current role of the Robot. The generic *ProcessWorkpiece* behavior looks up the agent's role in the corresponding knowledge and compares it to the capabilities available to the robot. If the role is part of the capabilities, the workpiece is processed. After processing, the robot checks if there is a cart available to pick up the workpiece or waits for a cart to arrive. The cart must announce its availability by sending a "READY-FOR-PICKUP" message to the robot. When the cart has arrived, the *GIVEWorkpieceToCart* behavior (*GIVE*) is instantiated to transmit the workpiece to the cart. After the protocol has finished, the loaded cart proceeds to the next way point on its route and the robot is ready to receive another workpiece. Note, that in this implementation communication is asymmetric. Carts are always the actors, who initially send messages. Choosing symmetric or asymmetric communication and deciding, which types of agents initiate communications is a domain specific design decision.

All agents can be reconfigured by the *VirtualCentralizedO/C*, which is also implemented as an agent in the following called O/C-agent. If one of the agents indicates a failure to the O/C-agent, a "BEGIN-RECONFIGURE" message is sent to all peers. On reception of such a message, the robot aborts its ongoing conversations and all currently performed tasks and sends its current configuration (i.e. the HAS

knowledge containing its capabilities) to the O/C-agent (this is done by a specific *RECONFIGURE* behavior). By accumulating all received configuration data, the O/C-agent can compute optimal solutions. Currently, the collected data is discarded and an alternate, predefined configuration is transmitted to the agents. Implementing different algorithms is current work (right now constraint solvers like ALLOY (Jackson, 2000) are under evaluation). As different reconfiguration algorithms do not have an influence on the implementation of the system (besides the O/C-agent), we will not go into more detail on computing *Role Allocations*.

After a *Role Allocation* has been computed, the the new *DO* knowledges are distributed among the agents. For the robot this means in most cases that it "switches its tool and changes the carts from/to which it takes/gives workpieces", e.g. from *Drill* to *Screwdriver*. If this happens, it is usually necessary to reroute the carts to reflect the new order of the robots. After the exchange is complete all agents reset themselves and indicate their ability to continue normal operation to the O/C-agent (implemented as *READY* behavior). As soon as the O/C-agent has received acknowledgments from all agents it resumes the application by sending a "BEGIN-OPERATION" message. *RECONFIGURATION* and *READY* behaviors are shared by all agent classes. The O/C-agent uses one behavior to handle the communication during a reconfiguration cycle and one to generate the new role allocation. The latter behavior is a stub and can be used to implement the algorithm for this task in the future.

Summary: The case study showed, that the generic implementation helps a lot for building Organic Computing systems. A broad class of systems can be conveniently designed using the Organic Design Pattern. The presented architecture gives a good guideline for implementation. Nevertheless some additional design decisions (like choice of communication) have to be made during this process. On the other hand this allows for more efficient implementations. Technically, we made the experience that AgentService is still a very young framework. In particular, memory leaks appear during runtime relatively frequently. As allocation and de-allocation of knowledges is handled very deeply within the AgentService runtime, we were not able to ultimately solve these problems. We are in touch with the AgentService developers in Genoa and it can be hoped, that these problems will be solved in the near future. We also connected the presented control system to a physical simulation of the adaptive production cell (using Microsofts Robotics Studio (Microsoft,)). This showed, that communication overhead (introduce by a multi-agent platform) is not an issue.

6 CONCLUSIONS

Organic Computing systems offer the door to a new generation of software controlled systems. Their self-X properties make them interesting for many domains where dependability is an important aspect. On the other hand, design, analysis, and construction of such systems is a difficult task. A useful aid in these project stages is the use of guidelines and patterns.

In this paper we presented a design pattern for organic computing applications and a method for implementation using C# and the multi-agent framework AgentService. It turned out that a generic implementation is possible and that it is of great help when building specific applications. The current state of work only allows for a centralized controller and is thus trimmed for medium scale applications, where the number of agents is limited (<100) and computation time is not scarce. In future work a decentralized controller using leader election seems not to difficult to implement.

When evaluating the implementation of the case study, some minor technical problems with AgentService's libraries (i.e. memory leaks) were found. For this reason the implementation of the case study took longer than expected and some restrictions to functionalities had to be applied. However, there is hoped that these problems will be solved with the next version of AgentService. It is also interesting to see how efficiently other case studies can be implemented with this generic implementation. Another open topic is to compare the presented C# implementation with a JAVA based generic implementation using the Jadex framework.

REFERENCES

Boccalatte, A., Gozzi, A., Grosso, A., and Vecchiola, C. (2004). Agentservice. In Maurer, F. and Ruhe, G., editors, *SEKE*, pages 45–50.

Boccalatte, A., Grosso, A., and Vecchiola, C. (2006). Implementing a mobile agent infrastructure on the .net framework. 4th International Conference in Central Europe on .NET Technologies.

Branke, J., Mnif, M., Müller-Schloer, C., Prothmann, H., Richter, U., Rochner, F., and Schmeck, H. (2006). Organic Computing – Addressing complexity by controlled self-organization. In *Proceedings of the 2nd International Symposium on Leveraging Applications of*

Formal Methods, Verification and Validation (ISoLA 2006).

FIPA ACL (2002). *FIPA ACL Message Structure Specification*. http://www.fipa.org/specs/fipa00061/.

FIPA Website (1996). FIPA, Foundation for Intelligent Physical Agents. http://www.fipa.org/.

Güdemann, M., Ortmeier, F., and Reif, W. (2006a). Formal modeling and verification of systems with self-x properties. In *Autonomic and Trusted Computing 2006, Proceedings*. Springer LNCS.

Güdemann, M., Ortmeier, F., and Reif, W. (2006b). Formal modeling and verification of systems with self-x properties. In Yang, L. T., Jin, H., Ma, J., and Ungerer, T., editors, *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. Springer.

Güdemann, M., Ortmeier, F., and Reif, W. (2006). Safety and dependability analysis of self-adaptive systems. In *Proceedings of ISoLA 2006, 2nd Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. IEEE CS Press.

Jackson, D. (2000). Automating first-order relational logic. In *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-00)*, volume 25, 6 of *ACM Software Engineering Notes*, pages pp. 130 – 139. ACM press.

Microsoft. Microsoft robotics studio developer center.

Müller-Schloer, C., von der Malsburg, C., and Würtz, R. P. (2004). Organic computing. *Informatik Spektrum*, 27(4):332–336.

Object Management Group, O. (2003). UML 2.0 OCL Specification.

Seebach, H., Ortmeier, F., and Reif, W. (2007). Design and Construction of Organic Computing Systems. In *Proceedings of the IEEE Congress on Evolutionary Computation 2007*. IEEE Computer Society Press. accepted for publication.