Guiding Organic Management in a Service-Oriented Real-Time Middleware Architecture

Manuel Nickschas and Uwe Brinkschulte

Institute for Computer Science
University of Frankfurt, Germany
{nickschas,brinks}@es.cs.uni-frankfurt.de

Abstract. To cope with the ever increasing complexity of today's computing systems, the concepts of organic and autonomic computing have been devised. Organic or autonomic systems are characterized by so-called self-X properties such as self-configuration and self-optimization. This approach is particularly interesting in the domain of distributed, embedded, real-time systems. We have already proposed a service-oriented middleware architecture for such systems that uses multi-agent principles for implementing the organic management. However, organic management needs some guidance in order to take dependencies between services into account as well as the current hardware configuration and other application-specific knowledge. It is important to allow the application developer or system designer to specify such information without having to modify the middleware. In this paper, we propose a generic mechanism based on capabilities that allows describing such dependencies and domain knowledge, which can be combined with an agent-based approach to organic management in order to realize self-X properties. We also describe how to make use of this approach for integrating the middleware's organic management with node-local organic management.

1 Introduction

Distributed, embedded systems are rapidly advancing in all areas of our lives, forming increasingly complex networks that are increasingly hard to handle for both system developers and maintainers. In order to cope with this *explosion of complexity*, also commonly referred to as the *Software Crisis* [1], the concepts of *Autonomic* [2,3] and *Organic* [4,5,6] Computing have been devised. While Autonomic Computing is inspired by the autonomic nervous system (which controls key functions without conscious awareness), Organic Computing is inspired by information processing in biological systems. However, both notions boil down to the same idea of having systems with *self-X properties*, most importantly *self-configuration*, *self-optimization* and *self-healing*. More specifically,

U. Brinkschulte, T. Givargis, and S. Russo (Eds.): SEUS 2008, LNCS 5287, pp. 90–101, 2008.
© IFIP International Federation for Information Processing 2008

- *self-configuration* means the system's ability to detect and adapt to its environment. An example for this property would be the plug-and-play found in modern computers, which is used to automatically detect and configure certain attached devices;
- *self-optimization* allows the system to autonomously make best use of the available resources, and deliver an optimal performance;
- *self-healing* describes the detection of and automatic recovery from run-time failures, for example by using heartbeat signals and restarting services that are not responding in time.

To present to the applications a homogeneous view on a distributed system of heterogeneous components, there usually is a layer called *middleware* on top of the components' individual operating systems, making the distributed nature of the system mostly transparent to the application developer. Within an organic computing system, we expect the middleware layer to autonomously achieve a high degree of transparency. This includes self-configuration even within a dynamic environment (such as found in ad-hoc networks), self-optimization at run-time, and self-healing in case of failures, thus providing a robust, efficient and uniform platform for the applications without human maintenance or intervention.

Another increasingly important requirement for today's distributed embedded systems is *real-time capability.* A real-time system must produce results and react to events in a timely, predictable manner, guaranteeing temporal restraints that are imposed by the applications.

In [7], we have proposed a service-oriented organic real-time middleware architecture that achieves self-X properties through a multi-agent-based approach. Services act as intelligent agents that use an auction mechanism to coordinate. These agents may move around within the system, finding optimal nodes to run on, and tasks are allocated to agents that are most appropriate. This approach handles self-organization and self-optimization. However, a mechanism needs to be devised that describes and defines dependencies between services, between resources and between tasks in order to provide guidance for task and resource allocation. This mechanism must also be able to describe properties of the hardware (such as attached sensors or actors or available resources). Since that kind of information is often domain specific, it is essential that the application or system designer be able to specify these properties without modifying the middleware, therefore a generic mechanism is needed. In this paper, we propose an approach for such a mechanism.

In Sect. 2 we mention related work. Section 3 gives an overview about our own architecture and motivates the need for guiding organic management in more detail. In Sect. 4 we describe our mechanism and analyze its properties. Sect. 5 shows how to combine that approach with our agent-based middleware architecture in order to achieve a sensible organic management, and Sect. 6 how to integrate it with the node-local organic management. Finally, in Sect. 7 we provide an example that demonstrates our ideas.

2 Related Work

In autonomic and organic computing, much research has been done in recent years (e.g. [8] for an overview). There are different approaches for implementing organic middlewares. The DoDOrg project [9] develops a digital organism consisting of a large number of rather simple, reconfigurable processor cells, which coordinate through an artificial hormone system. The OCμ middleware [10] features an observer-controller architecture with centralized organic managers. It targets smart office buildings with powerful, connected networks rather than embedded real-time systems. The service-oriented real-time middleware OSA+ [11] has a low footprint and is very scalable, thus it is particularly suitable for embedded distributed real-time systems. However, it does not feature self-X properties. The general architecture for an organic, service-oriented middleware inspired by the OSA+ approach has been developed in [12]. For this architecture, we described an agent-based approach for implementing self-X properties in [7], which uses concepts from multi-agent systems for coordination and task allocation. A short summary of our approach is given in Sect. 3. We are currently implementing and evaluating the proposed mechanism within the CAR-SoC project [13,14].

Other agent-based approaches for organic middlewares, such as [15,16,17], do not feature real-time capabilities. To our knowledge, a flexible, generic mechanism for describing service and resource dependencies in a service-oriented middleware has not yet been developed. Services in OSA+ are fixed on the resources they manage, and tasks are dispatched globally. Other approaches use central planning or do not consider dependencies at all.

3 Overview and Motivation

For an organic middleware, a *service-oriented* architecture proves to be a good choice. For implementing self-configuration, self-optimization and self-healing, a modular concept is vital. In such an architecture, tasks are processed by *services*, which in most cases are not part of the middleware core, but run as independent, loosely coupled entities, leading to a *microkernel design*. We have proposed and explained in detail such a design in [7]; here, we will only give a short overview. We have chosen a microkernel-based approach primarily for *scalability*, *flexibility, reliability, recovery* and *extensibility*. In our proposed system, services are *intelligent agents* as defined in [18]. Task allocation is done using an auction mechanism based on *ContractNet* [19,20]. This mechanism uses cost/benefit calculations in order to determine the most suitable service agent for processing a given task. This approach is distributed (i.e. does not require central control, thus avoiding a single point of failure) and real-time capable.

In addition to influencing the task allocation mechanism by computing sensible cost/benefit functions, service agents can perform self-optimization by negotiating with other agents (potentially swapping or delegating already allocated tasks if this proves to be more optimal) or by migrating to another node of the

distributed system that is more suitable. Sandholm [21] has shown that an optimal task allocation can achieved in a *ContractNet* that allows re-allocation of tasks in certain ways. By periodically re-evaluating the current task allocation and agent distribution, and appropriate reactions, the system will also adapt to a changing environment. In addition, the middleware core can start or shut down service agents on particular system nodes as needed in order to improve scalability and optimize work load.

All this should happen autonomously, without human intervention or configuration. However, dependencies between tasks or between agents, the need for particular resources and hardware limitations on particular nodes restrict the configuration space for an agent. For example, if a task needs a certain resource locally, an agent can only offer to process it if it sits on a node that has that resource available. Or, a particular agent can only run on a node that has a certain hardware sensor attached. Or an agent might require another service running on the same node in order to perform certain functions or run at all. In order to define such restraints and dependencies, a mechanism is needed that guides the system's self-configuration in a way that does not require manual intervention after initial setup. In particular, the following properties are desirable:

– Many dependencies and restraints are application-specific. Thus we need a generic mechanism that is separated from the middleware implementation such that the application developer (or even the user) can define them as needed.
– The same is true for describing the hardware configuration. A node's operating system must be able to communicate its hardware setup (such as attached sensors and actors) to the middleware in a way that is flexible and extensible. It must not be necessary to recompile or reconfigure the middleware if e.g. a new type of hardware device is available; an application service that supports this device should be able to recognize its existence and to make use of it without explicit support by the middleware.
– The mechanism needs to be real-time capable.
– The mechanism should be transparent to the application. In particular, it should not matter for the application *where* (on which node and by which service agent) a task is executed, as long as it is executed at all. Of course, the application needs to be able to specify the requirements for processing a task.

In the following sections we propose and describe a mechanism that has these properties.

4 A Capability-Based Mechanism for Guiding Organic Management

The proposed mechanism is based on so-called *capabilities*. Roughly speaking, a capability c is a globally unique, possibly application-specific identifier representing a particular feature, ability or resource. More formally, we have a set C

containing all known capabilities, hence $c \in C$. Most of the time we will consider *sets of capabilities*, taken from the power set $\mathcal{P}(C)$. Furthermore, on a given node, we have a set R of hardware resources (such as sensors or actors) and a set A of service agents. The subset $A^0 \subset A$ shall denote agents currently not running on the node, whereas $A^1 \subset A$ is the set of currently executing agents.

A hardware resource $r \in R$ provides a set $S_r^{prov} \in \mathcal{P}(C)$ of supported capabilities. A service agent $a \in A$, on the other hand, usually requires a certain set of capabilities $S_a^{req} \in \mathcal{P}(C)$ to run. Moreover, a running service agent might provide additional capabilities $S_a^{prov} \in \mathcal{P}(C)$ to the node it is executed on.

This allows the specification of dependencies between services, such that a service will only be started on a node if another service is already running on that node, or formally, an agent $b \in A^0$ can be started on the node if and only if

$$S_b^{req} \subset \left(\bigcup_{a \in A^1} S_a^{prov} \cup \bigcup_{r \in R} S_r^{prov} \right).$$

The management of capabilities of a node's resources and agents then boils down to performing set operations, and since we are targetting the real-time domain, we need to consider if those can be implemented efficiently. In particular, the middleware core needs to *join* sets and it needs to test if one set is a *subset* of another. For removing capabilities from sets, *subtraction* is needed.

A very time-efficient implementation represents capability sets by *bitstrings*, with each bit representing a given capability that is either present or not. In this case, the afforementioned set operations boil down to bit-wise logical operations that can be done efficiently in constant time. Let S and $T \in \mathcal{P}(C)$ be capability sets, and s and t the corresponding bitstrings. Then the following are equivalent:

Set operation	**Logical operation**
$S \cup T$	s OR t
$S - T$	s *AND NOT* t
$T \subset S$?	(s *AND* t) = t ?

If the core maintains a capability set containing all offered capabilities (by joining S^{prov} for all resources as well as started services), it can check if a given service agent can be started in constant time. Removing a service, however, can only be done in constant time if we can assume that a capability cannot be provided by more than one resource or agent; only then can we use set substraction to remove the provided capabilities from the global set. Otherwise, the core needs to check all remaining providers for that capability, so this operation needs linear time (in the number of resources and running agents).

One drawback of this mechanism is the fact that the global number of capabilities must be known beforehand (at compile time) in order to guarantee constant time operations; otherwise, one needs to provide for dynamically growing capability sets. Another drawback is that the representation of a capability set is not the most space efficient. If we have n capabilities in the system, we

need a bitstring of length n to represent a capability set regardless of the number of elements contained. A single capability, however, can be represented as an integer number and mapped to its corresponding bit using a list of bitmasks for set operations.

If the real-time constraints and hardware resources allow for a more complex implementation, one can also use a more dynamic approach, for example using a hierachical tree structure containing named capabilities, where each node acts as a namespace for its children. A capability is then described by its path starting from the root of the tree. The most prominent advantage of such an approach is that it is dynamically extensible; the number of known capabilities needs not to be known beforehand, and the use of namespaces allow for arbitrarily (application-specific) named capabilities without the risk of collisions. However, this data structure does not allow for constant-time processing of sets.

5 Combining the Capabilities-Based Approach with Service Agents

As summarized in Sect. 3, we have proposed a middleware architecture that realizes self-X properties using service agents that coordinate using an auction mechanism. Essentially, for a given task, an announcement is sent out to suitable agents within the system – where the suitability of an agent can be checked by comparing its set of provided capabilities to the task's set of required capabilities. Every suitable agent determines the cost processing the task would incur and sends this information to the core. The agent with the best offer gets awarded the task.

For this auction mechanism, it is vital that a service agent be able to compute a sensible cost/benefit function for processing the task. Such a function should consider the cost of using needed resources, and also capture quality-of-service parameters (such that the price for processing a task depends on the quality of the result). Thus, it makes sense to attach cost information directly to the capabilities. This means, that using a resource is mapped to "using" a capability, and the provider of that capability (e.g. the node operating system or another service agent) determines an appropriate cost value. Of course, the same is true for delegating subtasks to other agents, which also would be mapped to using the corresponding set of capabilities. Quality-of-service parameters can be attached to the cost inquiry. Thus, the total cost for processing a given task is composed of the cost of the needed resources and subtask processing, represented by the corresponding capabilities.

6 Integrating Capabilities with Node-Local Organic Management

Within the *CAR-SoC* project, we are currently implementing our proposed approach in a middleware we call *CARISMA*[1]. CAR-SoC aims to build a

[1] Connected Autonomous Real-time Intelligent Service-based Middleware Architecture.

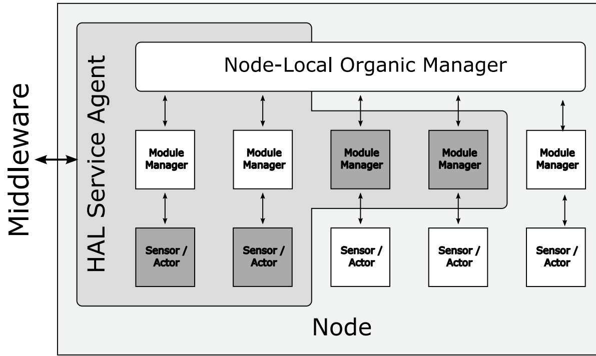

Fig. 1. Integration of node-local and global organic management. The shaded parts are components of the HAL Service agents, while white boxes belong to local organic management.

distributed embedded real-time system, with organic properties being employed throughout the whole stack. The individual nodes within the network run an operating system (called *CAROS* [22]) that features node-local organic management as described in [23]. We will not go into this concept in detail here, but only summarize the basic principles in a very concise and simplified manner as far as it concerns the interaction of the middleware layer's organic manager with the individual nodes (Fig. 1).

CAROS employs a two-staged organic management approach. On the lower level, small management units, so-called *Module Managers*, each manage a small set of system parameters, usually tied to a specific hardware or software module. Module managers receive raw monitoring data and can directly change the system parameters they manage. If a decision cannot be made on this lowest level, pre-interpreted monitoring data in a generic format is forwarded to the upper level, the node-local organic manager, which can make decisions based on node-wide status information.

As it turns out, this approach integrates very well with the capability-based approach for global organic management we propose here. A special middleware service agent (called HAL Agent, for Hardware Abstraction Layer) manages the interaction between the middleware core and the underlying node operating system. Its main task is to translate the specific hardware features into capabilities, and to attach a sensible cost/benefit function to using a given capability in order to influence the global organic management by way of the afforementioned auctioning mechanism. To accomplish that, the HAL agent registers itself as a module manager for the system parameters that can be monitored and/or influenced by the middleware layer.

On the middleware level, using a capability will generally require the usage of node resources. The module manager for a given resource attaches a *cost scale* to using that resource. In addition, system parameters might need to be adjusted. For example, using a capability might require a certain amount of processing

power, which in turn might require to increase the node's processor frequency. On the node level, any actor that changes system parameters also has a cost scale attached. Changing a parameter, or a combination of parameters, will improve or worsen the node's state; for example, increasing the processor frequency in order to offer required computing power also increases energy consumption and system temperature and therefore the overall cost value of the action. In order to influence the global organic management, the node's cost scales are integrated by the HAL Agent and attached to the capabilities to be used by the global auctioning mechanism. Summarizing this, the cost/benefit function for using a given capability is derived from both the cost scales for the needed resources and for changing the node's state.

On the other hand, the HAL Agent can also register its own actors on the node level, thus allowing the node's organic manager to actively influence global organic management. For example, one actor might be "move this service from this node to another". The attached cost scale would reflect the possible alternative locations for running the given service (obtained as the result of an auctioning round). Another possibility might be to change quality-of-service (QoS) parameters of a running service in order to improve the node's state; the cost scale the HAL agent provides for this actor would reflect the incurring quality degradation on a global level.

This shows that within the architecture proposed in the CAR-SoC project, both global and local organic management can interact in various ways. Capabilities as suggested here allow mapping global properties to local resources and system parameters and vice versa. This diversity in implementation of organic features will be very interesting to explore; in particular, how to fine-tune the balance between the global and local organic managers, since both levels can influence the other's decisions passively (by modifying cost values) or actively (by performing actions).

7 Example

For a better understanding, we will in this section discuss a simple example that demonstrates auction-based task allocation using capabilities in order to provide self-X properties. For the sake of brevity and simplicity, we will only describe the global level of organic management and not consider the interaction with the node-local organic management as described in Sect. 6.

Consider the front lighting control of a car. We assume that this car has a left and a right headlight, a left and a right turn signal and a left and a right front fog light. These lights are controlled by two microcontrollers running our agent-based middleware (Fig. 7). As example scenario for self-X properties, consider that one of the turn signals breaks and can no longer be used. The system shall detect this failure and then autonomously decide to let the corresponding fog light blink in the future, since this behavior (signaling a turn with a fog light) is still safer than not signaling a turn at all. In addition, if the fog light also stops working, the system shall decide to use the headlight instead (which is

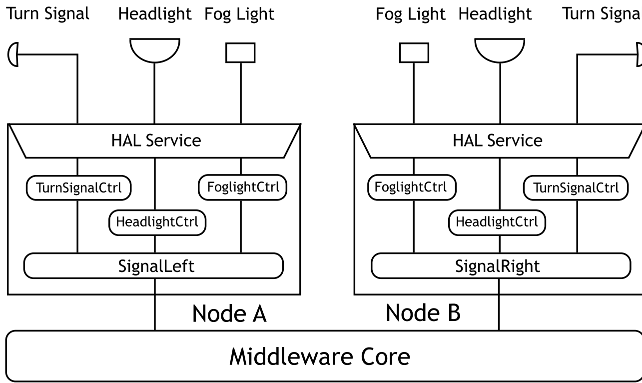

Fig. 2. Example scenario. Two controllers (Node A and B) each control three lights. Hardware is accessed through the HAL service providing appropriate capabilities. Control services make these available to higher-level, more complex services such as *SignalLeft* and *SignalRight*.

the worst method, but still better than nothing). This is an example for *self-healing*.

The system nodes provide capabilities (possibly through a hardware abstraction layer service HAL) that describe the hardware configuration. Each node has the capabilities HEADLIGHT, TURNSIGNAL and FOGLIGHT. In addition, the physical location of these lights needs to be defined, so the nodes have the capabilities RIGHT resp. LEFT. Each light is controlled by a service that allows to toggle its state and that monitors the function of the corresponding light. Thus, these services need the corresponding capability to run, and they provide the capabilities ON and OFF for their lamp. In addition, information about the nature of the light is provided. All lights can ILLUMINATE the road, and all lights can SIGNAL a turn. However, they are not equally suited for these tasks, so these capabilities carry a cost value[2] with them that describes how appropriate a job is for the given light. Here, application-specific knowledge is used in order to influence the organic management appropriately. Since a headlight can illumante the road quite well, the cost is 0; on the other hand, using it as a turn signal is only desired if it is the only working light left, so we associate a cost of 100 with that. A fog light can illuminate the road somewhat, but not very good. We give it a cost of 50. It also can signal a turn if necessary, and is preferred to the headlight for that task, so the cost for signaling should not be 0, but also less than 100. We give it a 50. The turn signal can hardly illuminate the road, but we still want to turn it on if the other lights are all broken; so the cost for illumination is 500. Of course, it signals a turn for free because it is the preferred device for that task. The services, capabilities and cost are summarized in Tab. 1.

[2] Note that the absolute number does not matter; it is the relation between different cost values that guides task allocation.

Table 1. System services in the example scenario, provided capabilities and associated cost. The left table shows the capabilities provided by the hardware configuration of the two nodes; the right table shows the services for controlling the lighting.

Service	Capability	Cost
Node A HAL	HEADLIGHT	0
	FOGLIGHT	0
	TURNSIGNAL	0
	LEFT	0
Node B HAL	HEADLIGHT	0
	FOGLIGHT	0
	TURNSIGNAL	0
	RIGHT	0

Service	Capability	Cost
HeadlightCtrl	ON	0
	OFF	0
	ILLUMINATE	0
	SIGNAL	100
FoglightCtrl	ON	0
	OFF	0
	ILLUMINATE	100
	SIGNAL	50
TurnSignalCtrl	ON	0
	OFF	0
	ILLUMINATE	500
	SIGNAL	0

7.1 Capabilities in Action

Now, services for signaling a right or left turn shall be started. To run, these services (named *SignalLeft* and *SignalRight*) need the capabilities SIGNAL and LEFT or RIGHT, respectively. Consider the service *SignalLeft*. The middleware core can only start this on Node A, which provides the necessary capabilities. If all lights are working, *SignalLeft* will use the SIGNAL capability provided by the *TurnSignalCtrl* service, since it is the cheapest. It can then offer the capability SIGNAL_LEFT with an attached cost of 0 to the application. Now assume that the blinker lamp breaks. *TurnSignalCtrl* will notice this and notify the system that it can no longer provide its capabilities. This information is forwarded to *SignalLeft*. It now tries to find another service that provides the SIGNAL capability; *FoglightCtrl* can do this for 50. This means that SIGNAL_LEFT is still being provided to the application, but for a cost of 50 now. Only if the fog light also breaks will *SignalLeft* resort to the headlight, since it is even more expensive to use. Should all lights on the left side be disabled, *SignalLeft* could no longer find the SIGNAL capability and would need to shut down. A similar scenario can be imagined for road illumination, causing the system to resort to the fog light if the headlight breaks, and still using the turn signal if all else is gone.

7.2 Summary

This example demonstrates how capabilities can be used to describe the hardware configuration of a system, and how an auction-based task allocation mechanism can make use of appropriately defined capabilities for realizing self-configuration and self-healing.

8 Conclusion

In this paper, we have presented a method for describing dependencies between services and resources in a service-oriented organic middleware. In such a middleware, self-X properties are realized by allocating tasks to the most suitable service, and executing services on the most suitable node. This information is often application-specific and cannot be hard-coded within the middleware. We have shown a generic mechanism based on capabilities that allows guiding the organic management. It is possible to specify both dependencies between services and dependencies on resources. By combining the capability mechanism with an auction-based task allocation mechanism as described in [7], self-optimization, self-configuration and self-healing can be achieved.

We are currently implementing the proposed mechanism in our middleware CARISMA, which is part of the CAR-SoC project [13]. This project extends the focus of organic computing to the hardware by developing an embedded hard-real-time system supporting autonomic computing principles. CARISMA closely interacts with the local (per-node) organic management to create a robust self-configuring, self-optimizing and self-healing distributed system. We have described how we plan to integrate both the global and node-local organic management by mapping capabilities and attached cost functions to local monitors and actors.

References

1. Gibbs, W.W.: Software's chronic crisis. Scientific American, 72–81 (September 1994)
2. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer, 41–50 (January 2003)
3. Horn, P.: Autonomic Computing: IBM's Perspective on the State of Information Technology. IBM Research, Armonk (October 2001)
4. Müller-Schloer, C., v.d. Malsburg, C., Würtz, R.P.: Organic computing. Aktuelles Schlagwort in Informatik Spektrum, 332–336 (2004)
5. Schmeck, H.: Organic computing – a new vision for distributed embedded systems. In: Proc. of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005), pp. 201–203. IEEE Computer Society, Los Alamitos (2005)
6. VDE/ITG/GI: Positionspapier Organic Computing: Computer und Systemarchitektur im Jahr 2010 (2003)
7. Nickschas, M., Brinkschulte, U.: Using multi-agent principles for implementing an organic real-time middleware. In: Proc. 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2007), Santorini, Greece, pp. 189–195. IEEE Computer Society, Los Alamitos (2007)
8. Deutsche Forschungsgemeinschaft: DFG SPP 1183 Organic Computing
9. Becker, J., Brändle, K., Brinkschulte, U., Henkel, J., Karl, W., Köster, T., Wenz, M., Wörn, H.: Digital on-demand computing organism for real-time systems. In: ARCS Workshops, GI. LNI, vol. 81, pp. 230–245 (2006)

10. Trumler, W.: Organic Ubiquitous Middleware. Ph.D thesis, Universität Augsburg (2006)
11. Picioroagă, F.: Scalable and Efficient Middleware for Real-Time Embedded Systems. A Uniform Open Service Oriented, Microkernel Based Architecture. Ph.D thesis, Université Louis Pasteur, Strasbourg (December 2004)
12. Nickschas, M.: Konzeption einer Anwendungsschnittstelle für eine echtzeitfähige Middleware mit Selbst-X-Eigenschaften. Master's thesis, Universität Karlsruhe (TH) (September 2006)
13. Uhrig, S., Maier, S., Ungerer, T.: Toward a Processor Core for Real-time Capable Autonomic Systems. In: Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology, December 2005, pp. 19–22 (2005)
14. Kluge, F., Mische, J., Uhrig, S., Ungerer, T.: Car-SoC – towards an autonomic SoC node. In: ACACES 2006, L'Aquila, Italy, July 2006, Academia Press, Ghent, Belgium (2006) (Poster Abstracts)
15. Kasinger, H., Bauer, B.: Combining multi-agent-system methodologies for organic computing systems. In: Proceedings of the 16th International Workshop on Database and Expert Systems Applications (DEXA 2005). IEEE Computer Society, Los Alamitos (2005)
16. Mamei, M., Zambonelli, F.: Self-organization in multi agent systems: A middleware approach. In: Engineering Self-Organising Systems, pp. 233–248 (2003)
17. Serugendo, G.D.M., Gleizes, M.P., Karageorgos, A.: Self-organization in multi-agent systems. Knowl. Eng. Rev. 20(2), 165–189 (2005)
18. Weiss, G. (ed.): Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence. The MIT Press, Cambridge (1999)
19. Smith, R.G.: The contract net protocol: High-level communication and control in a distributed problem solver. IEEE Transactions on Computers C-29(12), 1104–1113 (1980)
20. Sandholm, T.W.: An implementation of the contract net protocol based on marginal cost calculations. In: Proceedings of the 12th International Workshop on Distributed Artificial Intelligence, Hidden Valley, Pennsylvania, pp. 295–308 (1993)
21. Sandholm, T.W.: Contract types for satisficing task allocation: I Theoretical results. In: AAAI Spring Symposium Series: Satisficing Models, Stanford University, CA, March 1998, pp. 68–75 (1998)
22. Kluge, F., Mische, J., Uhrig, S., Ungerer, T.: An Operating System Architecture for Organic Computing in Embedded Real-Time Systems. In: Rong, C., Jaatun, M.G., Sandnes, F.E., Yang, L.T., Ma, J. (eds.) ATC 2008. LNCS, vol. 5060. Springer, Heidelberg (2008)
23. Kluge, F., Uhrig, S., Mische, J., Ungerer, T.: A two-layered management architecture for building adaptive real-time systems. In: Proceedings of the 6th IFIP Workshop on Software Technologies for Future Embedded & Ubiquitous Systems (SEUS 2008) (2008)