CARISMA – A Service-Oriented, Real-Time Organic Middleware Architecture

Manuel Nickschas and Uwe Brinkschulte
Institute for Computer Science
University of Frankfurt/Main, Germany
{nickschas, brinks}@es.cs.uni-frankfurt.de

***Abstract*—To cope with the ever increasing complexity of today's computing systems, the concepts of autonomic and organic computing (AC/OC) 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. The CAR-SoC project aims to realize such a system, employing AC/OC properties throughout the whole hardware and software stack. In this paper, we describe the architecture of our middleware CARISMA, which interconnects the individual CAR-SoC nodes. We show how middleware services can be treated as intelligent agents, and how we can use a multi-agent coordination mechanism for implementing the organic management, in particular self-configuration and self-optimization, in a decentralized and efficient way. We also elaborate on how to provide some guidance in order to take dependencies between services into account as well as the current hardware configuration and other application-specific knowledge. Integrating global organic management with the per-node local organic management is another issue that is presented in this paper. Last but not least, we provide some details regarding our ongoing implementation of CARISMA.**

I. 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]–[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,

- *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, even in a changing environment, which requires continuous adaption;
- *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; in [8], we have described a method to guide this mechanism in order to describe and define dependencies between services, resources and tasks. We are now implementing the suggested mechanisms within the *CAR-SoC* project [9], and explore the interactions between global (inter-node) and local (per-node) organic management. In this article, we present our approaches as applied to the concrete middleware *CARISMA*[1] we are developing within the *CAR-SoC* project [9]. We show how the various mechanisms for global (inter-node) and local (per-node) organic management interact, and we give some details about our ongoing implementation.

In Section II we mention related work. Section III gives a short overview of the CAR-SoC architecture. In Section IV we analyze the requirements for our system and draw conclusions for CARISMA's design

[1]Connective Autonomic Real-time Intelligent Service-oriented Middleware Architecture

doi:10.4304/jsw.4.7.654-663

and architecture. Sections V and VI describe our multi-agent-based concept for realizing task allocation. In Sections VII and VIII we present our mechanism for guiding organic management by specifying dependencies between services and resources, and analyze its properties. Section IX talks more about how to integrate CARISMA with the rest of the CAR-SoC stack, and Section X gives some details about specific issues in our current implementation.

II. Related Work

There is a plethora of different middleware systems today, such as CORBA [10], DCOM [11], .NET [12] or Java RMI [13]. While there are real-time extensions for some of them, none of them features organic aspects. On the other hand, in recent years much research has been done in the area of organic [6] and autonomic [2], [3], [14] computing; see [15] for an overview. The DoDOrg project [16] develops a digital organism consisting of a large number of rather simple, reconfigurable processor cells, which coordinate through an artificial hormone system. AMUN [17], subsequently enhanced as the OCμ middleware [18], is an interesting approach to an ubiquitous organic middleware, which features the main concepts of self-organization, self-healing and self-optimization. It uses an observer-controller architecture with centralized organic managers. It targets smart office buildings with powerful, connected networks rather than embedded real-time systems. However, all of these approaches do not consider real-time aspects, and their architectures are not suitable for supporting real-time applications. This is also true for the agent-based, organic middlewares that are described in [19]–[21].

The service-oriented real-time middleware OSA+ [22] 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 [23]. 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. We added a guidance mechanism for task allocation in [8]. To our knowledge, such 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.

We are currently implementing and evaluating these mechanisms within the CAR-SoC project [9], [24]. The CAR-SoC project as a whole is, to our knowledge, the only project that focuses on providing a complete organic, real-time stack, including the hardware itself (the *CarCore* processor [25]), a real-time organic per-node operating system (*CAROS* [26], [27]) and our middleware to manage a distributed system of such nodes.

III. The CAR-SoC Architecture – An Overview

The CAR-SoC[2] project pursues the exploration of how to apply principles of organic and autonomic computing to a real-time distributed system consisting of a new generation of high-performance embedded microcontrollers (*Systems on Chip*, SoCs). The paradigms of organic/autonomic computing are essential on all layers of the architecture. At the lowest level, an SMT microcontroller (called CarCore) is under development that allows for the scheduling of a large number of hardware threads; only the available memory limits the number of threads [25]. The overhead for thread-switching is rather small, because the time needed for context switches is used for processing other thread slots in the meantime. CarCore supports Guaranteed Percentage (GP) scheduling [28], which assigns a specific percentage of the available processor time to each of the hardware threads. This allows for the strict isolation of real-time threads. CarCore is binary compatible with Infineon's TriCore 2 processor, which means that existing toolchains can be used for development. CarCore's multi-threading capabilities in particular make running a concurrent architecture and the use of many helper threads feasible on both the operating system and the middleware layers.

Each SoC node is controlled by an operating system called CAROS [27]. It employs organic/autonomic computing on the node level in order to realize self-X properties for managing the hardware. It can, for example, tweak hardware parameters (such as the processor clock or scheduling parameters) to optimize the hardware configuration for the given load and other requirements. Helper threads perform organic management such as reconfiguration in the background without disturbing real-time threads. CAROS offers a POSIX-like interface for accessing devices and the organic manager in addition to providing a native interface that we can use for integrating our middleware. See Section IX) for more details about how organic management is implemented in CAROS and how we plan to integrate our middleware with it.

Our middleware CARISMA interconnects the individual nodes and presents a homogeneous view to the applications running on top of it (Fig. 1). CARISMA too makes extensive use of threading, as we will see in the following sections.

IV. Middleware Design and Architecture

In this section, we give an overview about the requirements for our middleware as well as describe some of the design decisions we made following from those.

A. Requirements

CARISMA is intended to run in a potentially large network of SoC nodes, interconnected by potentially unreliable means. Robustness even under these circumstances is a must. To help achieving that, the system as a whole and CARISMA in particular shall implement

[2]Connective Autonomic Real-time System-on-Chip

doi:10.4304/jsw.4.7.654-663

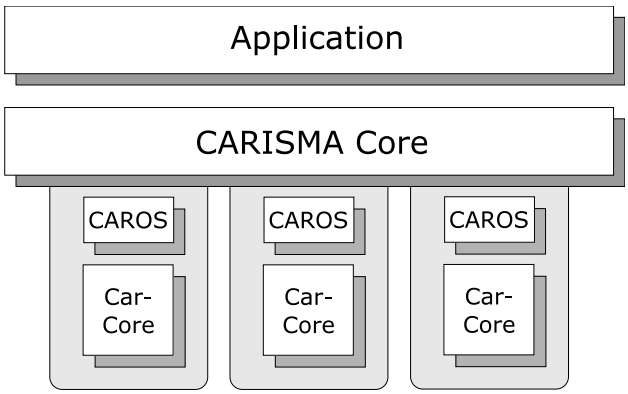

Figure 1. The CAR-SoC Architecture. On each node, an SMT microcontroller called *CarCore* is controlled by the system software *CAROS*. The nodes are interconnected by the middleware *CARISMA*, which provides a heterogeneous view to the applications running on top of the stack.

paradigms of organic/autonomic computing. Most prominent among the different *self-X properties* we want to support are *self-configuration*, *self-optimization* and *self-healing*. Furthermore, we require real-time capabilities. This means that the system needs to exhibit a predictable run-time behavior under normal conditions, including dynamic reconfiguration of components; but even facing partial breakdown of nodes, connections or other parts of the system, CARISMA needs to provide a best effort approach to maintain both functionality and real-time requirements as long as possible.

From these preconditions, we can already draw some basic conclusions for the basic middleware design, which we would like to summarize here:

- CARISMA must operate in a decentralized manner. We cannot allow for a centrally controlled coordination mechanism, nor for a *master node* or similar, because this would not scale for larger networks. It also would introduce a *single point of failure*, hence a much higher probability for suffering from a fatal breakdown even though only a small part of the system might be defective.
- We require a modular design, rather than a monolithic one. Components must be loosely coupled, in order for them to be easily restarted, replaced, migrated or upgraded without affecting the rest of the system. This is also crucial for robustness, since in a monolithic system, a failing module could take down the whole node. Additionally, modularity allows for flexible and scalable configuration.
- CARISMA does not live in a static environment. Nodes may come and go, new components may be added to the system, others might be removed or break down. Following the paradigms of *self-optimization* and *self-configuration*, partially also *self-healing*, we expect the middleware to adapt to such changes, striving to optimize its performance in any given situation, without human intervention. Again, we cannot rely on a central controller or monitor for this task; CARISMA needs to make decisions based on local information only.
- Communication (and thus, task processing) must happen asynchronously. Synchronous operation is only viable if communication lines have a very low latency, and it does not scale for larger networks and/or slower connections. In addition, a failing connection or recipient could cause the sender to block for a long time, which is not desirable.
- As mentioned in Section III, the hardware CARISMA is intended to run on supports a large number of real-time threads each with a guaranteed share of the available processing power. This allows for a modular design with concurrent processing of tasks and jobs, and it is a feature that should and will be reflected in the middleware's design.

B. Basic Middleware Design

For reasons that will become clear in the following sections, we have opted for a *service-oriented* architecture. One example for a service-oriented real-time middleware CARISMA shall be inspired by is OSA+ [22]. While not featuring organic computing principles, this middleware has proven to be efficient, scalable and real-time capable, and its architecture is suited for a setting which we also target CARISMA to. Some work towards implementing organic management in OSA+ has been done [29], but the framework turned out to be not flexible enough for our needs. One of the major design flaws in the earlier approach is the need for a global organic manager with a global world view, which clearly violates the requirements for CARISMA. Extending the OSA+ framework with the approach we envision for CARISMA would not be practical. In addition, CARISMA needs to function within the CAR-SoC framework; thus we decided to write CARISMA from scratch, while keeping an eye on some of the design principles in OSA+.

In a service-oriented architecture such as OSA+, most of the middleware's as well as the applications' functionalities are implemented as a number of services running on a *microkernel* whose main tasks are basic service management and communication. The service interface is accessed, in a generic way, via *jobs.* A job consists of an order, together with information about how and when the order should be executed, and a result. These jobs are sent between services running on the same or different nodes. They are assigned a globally unique identifier, so that longer-running jobs can be located and addressed (for example for requesting a status update of, or for providing new information to a job that is currently being processed).

The interface between an application and the middleware (*Application Programming Interface, API)* can be designed in a very simple way. System calls as found in traditional operating system designs can almost always

doi:10.4304/jsw.4.7.654-663

be mapped to a sequence of jobs[3] that is sent to the middleware core, which internally dispatches the jobs to appropriate services running on suitable nodes. The execution results are then sent back to the application. By making the service interface as universal as possible, many properties of the underlying hardware can be abstracted away from the application layer, including the fact that the application is running on a distributed system. In particular, all that matters to the application is *that* a given job is carried out obeying the given restraints, but not *how* or *where*. These decisions should be made autonomously by the middleware instead, increasing the configuration space which the middleware's organic manager can utilize to optimize the system's performance.

C. Communication

In a system consisting of loosely coupled components distributed over several hardware nodes, such as CARISMA, communication between the components cannot generally be implemented using normal method calls or shared memory. Often, distributed systems use *Remote Procedure Calls* (RPCs) or *Remote Method Invocation* (RMI) for communication between remote components. However, these mechanisms are synchronous, which means that the sender blocks until the receiver has finished processing the request. Consequently, the receiver must be available and responsive at the time of sending, and the sender cannot predict the time for the call to be finished. This is not desirable for CARISMA, where we have hard real-time conditions, and need to cope with failing services or nodes.

A more flexible alternative to RPCs/RMIs is message-oriented communication. *Message Passing* allows for supporting both transient and persistent, and both synchronous and asynchronous communication primitives [30]. In effect, all communication happens via messages, i.e. data structures following a common format, that are exchanged between sender and receiver. Such messages can encapsulate jobs as well as other types of requests. An expiration time ensures that messages will be discarded when they are no longer needed; for example, a job can be safely discarded if its deadline is already past due.

Messages allow for prioritized communication, another important feature of real-time systems to ensure end-to-end priorities; for this, they are stored in priority queues which can be implemented efficiently (see Section X-A for more details regarding the implementation). Services can opt for both polling and pushing of messages, depending on their needs. Queue maintenance, i.e. delivering new messages and discarding obsolete ones, can be done asynchronously by helper threads.

[3]It is also possible to move this mapping partly or completely into the middleware core, such that the application can access relatively high-level functions that are then broken down into sequences of primitive jobs internally. The finer the granularity, the higher the flexibility for organic management.

V. Service Agents

As mentioned previously, we aim at decentralizing organic management to avoid having a single point of failure. However, maintaining a consistent global state on every node is prohibitive. Since the topology of the distributed system is dynamic – new nodes can be added and existing ones removed (or go defunct) at any time – providing a global world view is not realistic. Transmitting all monitoring data from each node to every other – and keeping all the nodes in sync – would require considerable communication resources and management overhead. We therefore have to find ways to process most, ideally all monitoring data locally on the node where it is produced rather than transmitting it all to the node that decides where to dispatch a job to. Obviously, the services themselves are well suited to do this preprocessing:

- A service knows best which resources and monitoring data it needs to process a given job
- Services only need to know about locally available resources in order to assess if and how a job could be performed, because they cannot make use of remote resources anyway
- Thus, only using local data, a service can evaluate for a given job if it can be carried out at all, and if so, which quality can be achieved and what the costs are
- The CAR-SoC architecture encourages a concurrent, multithreaded approach; allowing each service to run in one or even multiple separate threads does not cause too much overhead scheduling-wise

In other words, a service can do a cost/benefit calculation for a given job based exclusively on locally available data. Only the result of this calculation needs to be compared to that of other services on other nodes in order to find out which service is best suited to handle a job. Therefore, there is no need to make the complete state of the nodes globally available.

Finding such a cost/benefit function that works for our scenario is not a trivial problem. It must factor in required resources, achievable quality and side effects on the environment, such as an increased energy consumption that the execution of the job might incur. These factors are also interdependent; for example, achieving a higher quality of service might cost more resources, and the system needs to find a good balance between contradicting factors. Finding a good metric to combine these different inputs into a comprehensive output is a major part of our ongoing research on this project, but out of the scope of this paper. For now, we will assume that a suitable cost/benefit function exists.

But not only the calculation of a cost/benefit value can be done within the services. Much of the organic management can be handled by the services themselves, rather than by an explicit organic management component, leading to the desired decentralization. The central idea of our approach is to not consider the middleware services as passive components doing a narrow task, but to be – to a certain extent – independent entities that

doi:10.4304/jsw.4.7.654-663

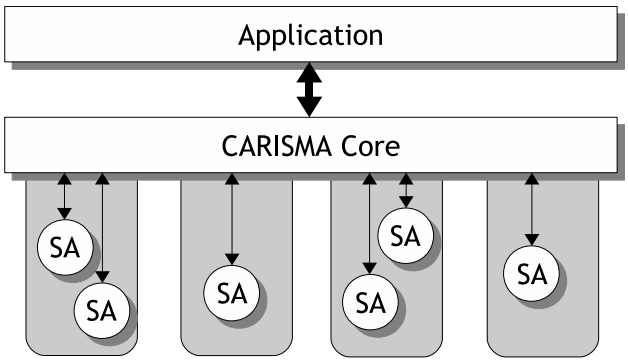

Figure 2. Middleware architecture with service agents. The agents run on several hardware nodes (shown as grey rectangles), which are logically connected by the middleware core. The application as well as the service agents always communicate through the core, never directly.

compete for available jobs, trying to optimize their own performance within the local environment of the node they are running on. By doing this, the services also improve the performance of the whole system.

Because CARISMA's services exhibit a certain intelligence and act independently, we call them *service agents*, and we consider our middleware a *multi-agent system* (Figure 2). The agents in this system compete for jobs the applications need to have processed. Therefore, the only role the middleware core plays in organic management is providing the infrastructure for agent coordination, whereas all configuration decisions are made by the services themselves. We will use a well-known multi-agent coordination mechanism in order to implement an optimal job allocation strategy.

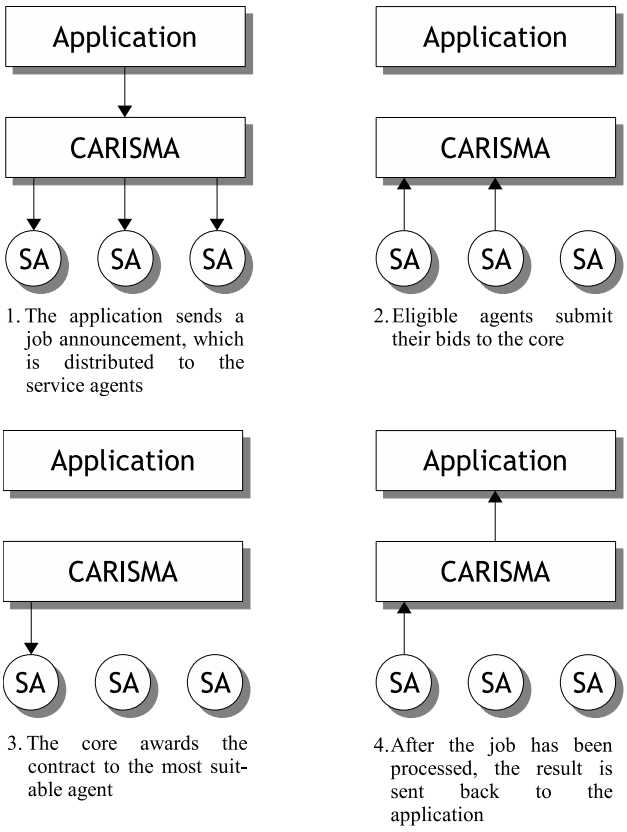

Figure 3. Job allocation scheme. The first three steps describe the auctioning mechanism. In the fourth step, the result of processing the job is sent back to the application. Not shown here is the (optional) recontracting of jobs between agents, which may happen after the initial allocation to optimize longer running tasks.

VI. Using Contract Net for Job Allocation

For our system, we need a real-time task allocation mechanism that gives us results quickly and with a deterministic upper time bound, but also allows for run-time optimization. A generic, simple and intuitive mechanism that can be used for allocating tasks is an *auction*. Auctions are often used in other (non-computer science) domains as well whenever something needs to be contracted out. In our case, the application offers jobs that it needs to have carried out, and suitable services use the afforementioned cost/benefit calculation to determine a "price" for performing the given job within the requested restraints. The service which can do the task for the lowest cost is awarded the job.

Within the domain of multi-agent systems, an auctioning mechanism called *Contract Net* [31], [32] is a well-researched high-level protocol for distributed negotiation. We have evaluated this protocol and later extensions, and its application to our special case of allocating jobs to service agents, already in detail in [7]. Here, we give only a short overview of the actual mechanism as it is employed in CARISMA. The basic cycle of the allocation of a job is depicted in Figure 3.

A few things are important to note, and we will summarize them here. First of all, this mechanism is suitable for a real-time environment. By choosing sensible deadlines or expiration times, one can ensure that a negotiation cycle is completed after a given amount of time. If the deadline for collecting bids has been reached, the middleware chooses the service agent with the best offer at that time, and discards bids that arrive after that. Contract Net is an *anytime algorithm*, meaning that once a valid state has been reached (i.e., at least one bid has arrived), waiting longer will only improve the situation, but never worsen it. The algorithm thus can wait as long as the deadlines allow, and return with the best result until then any time. If no bid arrives in time, the middleware returns a failure to the application, which can then handle the situation.

This mechanism is an example for *self-configuration*, because if at all possible within the given constraints, it will find a working job allocation scheme. It is also example for *self-optimization*: A longer running job can be recontracted again, and if the benefit of moving to another service agent (possibly on another node with better resource availability) outweighs the cost of migration, the job can be reallocated, thus improving the state of the system as a whole. Recontracting or delegating jobs can also be enforced by changing conditions, such as a node's battery running low. As shown in [33], Contract Net can reach a global optimum if auctioning between multiple agents as well as of several jobs at once is allowed. By

doi:10.4304/jsw.4.7.654-663

periodically re-evaluating the current task allocation and agent distribution, and appropriate reactions, the system will 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.

VII. Guiding Organic Management: Capabilities

Task allocation as described in the previous section should happen autonomously, without human intervention or configuration. However, the service agents need some guidance, because dependencies between tasks or between agents or 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 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.
- The mechanism needs to be real-time capable.

Our approach for guiding organic management in CARISMA is based on what we call *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
$S \cup T$	$s \vee t$
$S - T$	$s \wedge \neg t$
$T \subset S$?	$(s \wedge t) = t$?

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). Because such a clean-up operation can be handled by a helper thread in the background, this does not pose a problem.

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

doi:10.4304/jsw.4.7.654-663

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 could 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. Because CARISMA is intended to support hard real-time and operates within an embedded system with limited resources, we have opted for the bitstring implementation, and we will revisit this issue in Section X-C.

VIII. Combining Capabilities With Service Agents

For the auction mechanism employed in CARISMA, it is vital that a service agent be able to compute a sensible cost/benefit function for processing a 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. CAROS 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. Therefore, 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.

A. Example

A short example shall clarify how capabilities work. Consider the front lighting of a car. A car has headlights, turn signals and foglights. Suppose that each individual light is controlled by a service agent that can turn it on and off, and hence provide blinking and steady lighting. Each light can illuminate the road or signal a turn in the direction of its location;[4] but obviously, the quality a given light can achieve for both tasks varies. For illuminating the road, one would of course use the headlights. But what happens if they break? Then it would still be better to use the foglights rather than to not illuminate the road at all. If both head- and foglights break down, we could still use the turn signals for at least a little bit of light. Similarly it works for signaling a turn; if a turn signal does not work anymore, we would prefer using the foglight instead over not signaling at all, and so on.

In this scenario, all service agents controlling a light can both illuminate and signal to various degrees; hence, all service agents offer both these capabilities to the lighting control application. By attaching varying costs to using them, however, they can influence the task allocation in an optimal way: If the headlight agent offers the "illuminate" capability for the lowest price, the middleware will always choose it for that task, until the agent breaks down and is no longer available. In this case the next-cheapest agent will be chosen, which should be the foglight agent, and so on.

This example shows how the task allocation within CARISMA can be influenced by the service agents, and ultimately the application developer designing the cost scales for using capabilities, without the need to change anything in the middleware core or the auctioning mechanism. Thus, our requirement for the mechanism to be generic rather than needing application-specific details is met.

IX. Integrating CARISMA and CAROS

It turns out that capabilities also play a most important role in integrating global organic management, i.e. CARISMA's service agents, with node-local organic management within CAROS. We will now see that CAROS can influence global organic management by using capabilities as a lever between itself and the middleware.

The organic management implemented in CAROS has been described in detail in [26], and we will 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. 4).

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 in CARISMA. A special service agent (called HAL agent, for Hardware Abstraction Layer) manages the interaction between CARISMA's core and CAROS. Its main task is translating the specific hardware features into capabilities, and to attach a sensible cost/benefit

[4]Location (i.e. left or right side of the car) can be described by capabilities as well, but for the sake of brevity, we ignore this here

doi:10.4304/jsw.4.7.654-663

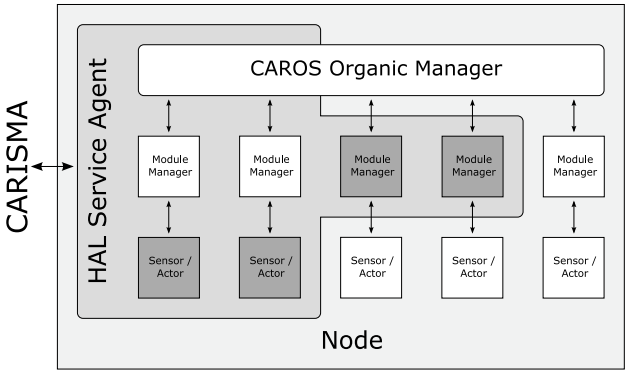

Figure 4. Integration of node-local and global organic management. The shaded parts are components of the HAL service agent, while white boxes belong to local organic management.

function to using a given capability in order to influence the global organic management by way of the aforementioned 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 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).

X. Implementation

CARISMA is currently in the implementation phase. We are limited to using C, because the TriCore 2 toolchain we use in CAR-SoC does not provide a predictable runtime behavior for C++, which would be needed for hard real-time. System functionality is provided by CAROS, which offers an interface similar to (a subset of) POSIX.

For the initial phases of development, we have written a high-level (CAROS API compatible) simulator that allows for simulating multiple networked nodes with different configurations that can be changed at run-time (for example for simulating defects). The underlying hardware and CAROS itself are not simulated however; we currently work with per-node and per-service capabilities that represent virtual hardware. Hence, we cannot use this simulator for evaluation of real-time capabilities or real-world hardware issues. As of now, we haven't fully implemented organic management and the HAL service, both of which rely heavily on the actual hardware and the organic management of CAROS. We have, however, started porting our code to the CarCore SystemC model running an actual CAROS instance, which will allow for the evaluation of CARISMA's real-time behavior as well as the integration of global and per-node organic management. For now, we would like to highlight some details of our current implementation.

A. Job and Message Handling

As explained in Section IV-B, CARISMA uses *Message Passing* for communication between service agents and the core. Jobs are special messages sent between the core and service agents; other message types include asynchronous invocation of service functions by the middleware core, or vice versa. All message types use a standard data format, including fields for the (globally unique and locatable) sender and intended receiver, optionally a job or transaction ID reference (such that a message can be referred to a job that is currently processed by the receivers), and a priority. A message can carry an arbitrary payload.

Messages are stored in and delivered through priority queues. Our priority queue implementation assumes that the maximum number of elements stored in such a queue is known and fixed at its creation; the necessary space is then preallocated. This is a requirement in a real-time setting, and also it allows us to use an efficient heap-based implementation. Inserting a message, removing the message with highest priority and increasing an arbitrary

doi:10.4304/jsw.4.7.654-663

stored message's priority all operate in $O(\log n)$, where n is the number of stored elements. This allows us to provide a sensible upper time bound for processing message queues.

The middleware core on each node maintains a global (node-wide) priority queue for incoming messages, and one additional queue for every running service, which stores its yet unprocessed messages (a message inbox). Services can post messages to the local CARISMA core; they are first stored in a node-wide priority queue. A separate helper thread processes the messages in order of priority. If the receiver resides on the same node, the message is forwarded to the receiver's own message queue (which is a priority queue as well). If the receiver resides on a different node, the message is sent away to the receiving node. If a message could not be delivered after reaching its expiration time, it will be removed from the queue and discarded, possibly generating an error reply.

B. Service Management

One of the main tasks of CARISMA's core is basic service management, i.e. localizing, loading, running and terminating service agents, delivering messages between them, and so on. To make implementing service agents as straightforward as possible, it is desirable that there be a common interface or skeleton for accessing a service's interface. Since we are limited to C, we cannot use object-orientation. However, CAROS supports loading shared libraries called *modules* at run-time. Such modules can have their own symbol namespace, a feature we use to avoid symbol collisions between services. This allows us to define some standard functions every service agent needs to provide, such as *init()* or *run()*, as well as access functions for the sets of required and provided capabilities. On loading of a service module, the core resolves those symbols and stores the resulting pointers in a data structure together with some more metadata about that service. For functions a service does not provide, a default implementation is used instead, which avoids code duplication for common functionality.

Throughout the system, a service can be identified and all its information accessed via a pointer to that data structure, hence such a pointer is used as a *service handle* whenever a service needs to be addressed. This avoids costly copying of data. Because all needed symbols are resolved on load (during a phase where all needed resources are allocated too, which cannot be done in real-time in any case), subsequent accesses to the service agent's methods are fast and predictable.

CAROS modules can be executed in a separate thread. The default implementation for the main function of a service agent provides an event loop that blocks until a message arrives, and calls the appropriate message handler on receipt. This kind of event-based programming avoids the need for synchronization and hence is very suitable in a real-time context, since the middlware core will never actively interrupt a running service.

Of course, the tradeoff for not using interrupts is that timely processing of incoming messages cannot be guaranteed, in the sense that the middleware core cannot usually know if the receiving agent is currently busy with processing other tasks rather than checking its message queue. Therefore, another mode of operation are passive services, which do not have their own event loop (or even their own thread), but rather provide hooks that are actively called by the middleware core. This is more lightweight, but also introduces issues with concurrency and real-time behaviour. A hybrid variant is a service agent allowing interrupts while featuring an event loop and active behavior. It depends on the needs of the application which of these possibilities should be used for a given service.

C. Capabilities

As described in Section VII, capability sets can be implemented in a time-efficient (albeit not necessarily space efficient) manner using bitstrings, where each bit position represents the presence of a given capability. In CARISMA, the size of capability sets is a global constant that needs to be set at compile time. A single capability, however, is represented by an integer number. The mapping between this integer and the corresponding bit in the set is done using a lookup table of bitmasks. A capability can be added to, removed from or tested for in a capability set in constant time. In particular, this allows for quick checking if a given service can run on a given node, or if a given job can be processed by a given service. Thus, we can check the necessary preconditions quickly before invoking the more costly auctioning mechanism.

XI. Conclusion and Future Work

In this paper, we have presented our middleware CARISMA, which is part of the CAR-SoC project [24]. 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 use an agent-based approach, using intelligent service we call *service agents*, for providing organic management on the middleware level. For implementing decentralized task allocation, we use an auctioning mechanism based on *Contract Net*, a coordination mechanism well known in the realm of multi-agent systems. We can describe dependencies between service agents and resources in a generic way by using what we call *capabilities*. We are integrating both the global and the per-node organic management by mapping such capabilities and attached cost functions to local monitors and actors.

Besides furthering the implementation of CARISMA, including porting it to and evaluating it on CAROS running on a SystemC simulator (and finally on real hardware), there are still several interesting aspects to explore. One of the most interesting issues requiring future

doi:10.4304/jsw.4.7.654-663

research is the mapping of resource requirements to a cost scale that allows for sensible organic management. This work mostly influences the design and implementation of the HAL agent that acts as a bridge between the locally running CAROS instance and the middleware. It also is central to the integration of both levels of organic management.

Other aspects of ongoing and future research include the communication between several nodes and the evaluation and possible enhancements of the auctioning mechanism, such as inter-agent and multi-job renegotiations. Transaction management and redundancy are important for providing robustness and the possibility of implementing certain aspects of self-healing, such as transparent recovery from failures. Last but not least, the efficient migration of running service agents between nodes, in particular such containing large state information, is a non-trivial issue that requires further research.

REFERENCES

[1] W. W. Gibbs, "Software's chronic crisis," *Scientific American*, pp. 72–81, Sept. 1994.

[2] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, pp. 41–50, Jan. 2003.

[3] P. Horn, *Autonomic Computing: IBM's Perspective on the State of Information Technology*. IBM Research, Armonk, NY, Oct. 2001.

[4] C. Müller-Schloer, C. v.d. Malsburg, and R. P. Würtz, "Organic computing," *Aktuelles Schlagwort in Informatik Spektrum*, pp. 332–336, 2004.

[5] H. Schmeck, "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'05)*, pp. 201–203, IEEE Computer Society, 2005.

[6] VDE/ITG/GI, "Positionspapier Organic Computing: Computer und Systemarchitektur im Jahr 2010," 2003.

[7] M. Nickschas and U. Brinkschulte, "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'07)*, (Santorini, Greece), pp. 189–195, IEEE Computer Society, 2007.

[8] M. Nickschas and U. Brinkschulte, "Guiding organic management in a service-oriented real-time middleware architecture," in *Proc. of the Sixth IFIP WG 10.2 International Workshop (SEUS 2008)* (U. Brinkschulte, T. Givargis, and S. Russo, eds.), Software Technologies for Embedded and Ubiquitous Systems, pp. 90–101, Springer, Oct. 2008.

[9] F. Kluge, J. Mische, S. Uhrig, and T. Ungerer, "Car-SoC – towards an autonomic SoC node," *L'Aquila, Italy ACACES 2006 Poster Abstracts*, July 2006. Academia Press, Ghent (Belgium).

[10] Object Management Group, *Common Object Request Broker Architecture: Core Specification*, version 3.0.3 ed., Mar. 2004. Abruf: März 2006.

[11] G. Eddon and H. Eddon, *Inside Distributed COM*. Microsoft Programming Series, Redmond, WA, USA: Microsoft Press, 1998.

[12] D. S. Platt, *Introducing Microsoft .NET*. Redmond, WA, USA: Microsoft Press, 2001.

[13] Sun Microsystems, Inc, "Java remote method invocation documentation," 2004.

[14] IBM, "Autonomic computing." Online Resource. Accessed: March 2006.

[15] Deutsche Forschungsgemeinschaft, "DFG SPP 1183 Organic Computing."

[16] J. Becker, K. Brändle, U. Brinkschulte, J. Henkel, W. Karl, T. Köster, M. Wenz, and H. Wörn, "Digital on-demand computing organism for real-time systems," in *ARCS Workshops*, vol. 81 of *LNI*, pp. 230–245, GI, 2006.

[17] W. Trumler, J. Petzold, F. Bagci, and T. Ungerer, "AMUN: An autonomic middleware for the smart doorplate project," *Personal Ubiquitous Comput.*, vol. 10, no. 1, pp. 7–11, 2005.

[18] W. Trumler, *Organic Ubiquitous Middleware*. PhD thesis, Universität Augsburg, 2006.

[19] H. Kasinger and B. Bauer, "Combining multi-agent-system methodologies for organic computing systems," in *Proceedings of the 16th International Workshop on Database and Expert Systems Applications (DEXA'05)*, IEEE Computer Society, 2005.

[20] M. Mamei and F. Zambonelli, "Self-organization in multi agent systems: A middleware approach.," in *Engineering Self-Organising Systems*, pp. 233–248, 2003.

[21] G. D. M. Serugendo, M.-P. Gleizes, and A. Karageorgos, "Self-organization in multi-agent systems," *Knowl. Eng. Rev.*, vol. 20, no. 2, pp. 165–189, 2005.

[22] F. Picioroagă, *Scalable and Efficient Middleware for Real-Time Embedded Systems. A Uniform Open Service Oriented, Microkernel Based Architecture*. PhD thesis, Université Louis Pasteur, Strasbourg, Dec. 2004.

[23] M. Nickschas, "Konzeption einer Anwendungsschnittstelle für eine echtzeitfähige Middleware mit Selbst-X-Eigenschaften," Master's thesis, Universität Karlsruhe (TH), Sept. 2006.

[24] S. Uhrig, S. Maier, and T. Ungerer, "Toward a Processor Core for Real-time Capable Autonomic Systems," in *Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology*, pp. 19–22, Dec. 2005.

[25] J. Mische, S. Uhrig, F. Kluge, and T. Ungerer, "Carcore: A smt microcontroller with virtually unbounded thread number." Zur Veröffentlichung vorgesehen, 2008.

[26] F. Kluge, S. Uhrig, J. Mische, and T. Ungerer, "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.

[27] F. Kluge, J. Mische, S. Uhrig, and T. Ungerer, "An Operating System Architecture for Organic Computing in Embedded Real-Time Systems," in *Proceedings of the 5th International Conference on Autonomic and Trusted Computing (ATC-08)*, (Oslo, Norway), Springer, Jun 2008.

[28] U. Brinkschulte, J. Kreuzinger, M. Pfeffer, and T. Ungerer, "A scheduling technique providing a strict isolation of real-time threads," in *Seventh IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2002)*, Object-Oriented Real-Time Dependable Systems, Jan. 2002.

[29] M. Pacher, A. von Renteln, and U. Brinkschulte, "Towards an organic middleware for real-time applications," in *Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2006.

[30] A. S. Tanenbaum and M. van Steen, *Distributed Systems. Principles and Paradigms*. New Jersey: Prentice Hall, 2002.

[31] R. G. Smith, "The contract net protocol: High-level communication and control in a distributed problem solver," *IEEE Transactions on Computers*, vol. C-29, pp. 1104–1113, Dec. 1980.

[32] G. Weiss, ed., *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*. Cambrigde, MA: The MIT Press, 1999.

[33] T. W. Sandholm, "Contract types for satisficing task allocation: I Theoretical results," in *AAAI Spring Symposium Series: Satisficing Models*, (Stanford University, CA), pp. 68–75, Mar. 1998.

doi:10.4304/jsw.4.7.654-663