Self-configuration and Self-healing in AUTOSAR

Wolfgang Trumler, Markus Helbig, Andreas Pietzowski, Benjamin Satzger, Theo Ungerer
Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme
Institut für Informatik
Universität Augsburg
Eichleitnerstr. 30
D-86830 Augsburg, Germany
{trumler, pietzowski, satzger, ungerer}@informatik.uni-augsburg.de

ABSTRACT

State-of-the-art automobiles contain up to 70 microcontrollers. About 35% of all failures are caused by the electronic systems, either hardware or software. Furthermore the stockkeeping of spare parts is a cost intensive issue. On the other hand, microcontrollers evolve so fast that the used microcontrollers cannot be replaced by newer controller generations without any side effects.

AUTOSAR [4,5] is a first step to overcome these limitations in the design and maintenance of cars. Based on the AUTOSAR architecture we propose an organic middleware, which adds self-configuration and self-healing capabilities. We implemented a simulator to evaluate the self-configuration and self-healing and we present the evaluation results for different network sizes and a failure rate of up to 60%. Our results show an excellent performance for the self-configuration in terms of the amount of messages and that the self-healing is able to recover from failures as long as enough resources are available.

INTRODUCTION

Starting from a purely mechanical system, cars evolved to highly complex environments with a huge amount of embedded microcontrollers and a lot of software. Modern high-class automobiles employ up to 70 control units and more than 150 MB of software. Many of the functions that stem from these electronic systems help to reduce the pollution of the environment and increase the passengers’ security. The large part of the driver assistance and comfort functions would not be possible without these microcontrollers, sensors, and actuators. The majority of the innovations and improvements in a car are possible because of the availability of high-performance microcontrollers.

The drawback of the introduction of microcontrollers and software is that they are subject to errors because the absence of errors cannot be proven but only their presence. Current statistics [1] show that about 35% of all failures in cars stem from the electronic systems from both, hardware and software. Prognoses [2] say that there will be up to 100 networked control units in future cars, which might further aggravate the situation. On the other hand reducing the amount of microcontrollers to a minimum to avoid these potential sources of errors would be also the wrong way, because microcontroller performance is crucial for the development of more secure and comfortable cars.

The introduction of microcontrollers in the automobile has a second drawback one would not expect at first glance. The innovation cycle of microcontrollers is many times shorter than the lifetime of a car. During an expected lifetime of 12 years for a car, at least 3 to 5 microcontroller generations emerge. So one of the major problems is the stockkeeping of spare parts for the control units, which is expensive but cannot be avoided because new microcontrollers might have a different pin layout or functional behavior. Even if they are only changed internally, the new architecture might not support the old programs or the timing might have changed which can result in different external behavior. Much effort is needed to overcome these kinds of problems in the state of the art design and maintenance cycle.

A way out of this awkward situation must be found and it should overcome both the hardware as well as the software problems. There are efforts to address the problem of changing hardware platforms. One is the usage of the OSEK [3] operating system, which makes it much easier to develop software for different embedded automotive systems. Moreover the AUTOSAR Architecture [4,5] defines additional abstractions to further facilitate the design process and to add capabilities to map the software parts to different ECUs.
prior to the compilation step. In this way AUTOSAR also addresses some principal aspects of the former mentioned problems of hardware abstraction but does not offer any solution for the recovery from failures that are subject to hardware or software errors.

AUTOSAR lacks two principal requirements needed to build self-configuring and self-healing embedded systems: (1) AUTOSAR does not support a dynamic reconfiguration of the system at runtime and (2) the software components of AUTOSAR are bound to the microcontrollers as defined during the design cycle.

The problems related to a rising complexity are well known from other areas of computer science. Paul Horn, vice-president of IBM Research, postulated in 2001 a new self-management paradigm for future server systems, the Autonomic Computing [18]. About the same time the Organic Computing initiative [19] defined similar self-organization requirements for the next generation of distributed embedded systems. Both have in common that they describe the need for new systems to adopt life-like attributes, in particular self-configuration, self-optimization, self-healing, and self-organisation.

In this paper we propose a middleware approach to add self-configuration and self-healing capabilities to a system based on AUTOSAR. Two major changes in the design of embedded automotive systems are supposed. First, the current design of black boxes containing the parts (sensors, actuators, microcontroller, and software) required for a function must be sallied out. Second the software must be comprised of services instead of monolithic blocks. Together both requirements build the basis for a distributed and highly flexible hardware and software architecture.

The paper is structured as follows: the next section gives an overview of AUTOSAR as far as it is necessary for the understanding of the general architecture. Afterwards the proposed architecture of the Organic Middleware is described as well as the self-configuration and the self-healing mechanisms. Evaluations of our simulation environment will show the results gained from different setups. Related work is discussed in the homonymous section. The paper closes with a conclusion and a description of future work.

AUTOSAR OVERVIEW

The immense increase of embedded systems in automobiles raises new challenges in terms of interoperability and interconnection. Many functions, especially those concerning driver assistance, comfort, and multimedia functions need microcontrollers for complex calculations and control. Currently most of the suppliers deliver their functional units as a black box to hide the implementation details and to secure their knowledge.

OSEK Operating System [3] was a first step to build a common platform for the development of embedded systems for the automotive industry and to guarantee the interoperability of the assembled parts of different suppliers. The automotive manufacturers want to add more and more functionality into the car but they cannot afford to add an extra black box for every functional unit they get from a supplier. As a consequence functional unit that are purely software based need to fit into an architecture that can map software components to different microcontrollers. AUTOSAR [4,5] is supposed to facilitate this intention by introducing additional hardware abstractions and clear interfaces. In the following a short overview of the AUTOSAR architecture and the design flow is given.

ARCHITECTURE

The AUTOSAR architecture defines a middleware system for automotive applications. It will replace OSEK-OS in the future. The overall architecture is shown in figure 1. The topmost layer is the application layer where the software components (SW-C) run within the AUTOSAR Runtime Environment (RTE), which implements the Virtual Functional Bus (VFB). The interfaces of the software components have to be specified by the definition language of AUTOSAR. The AUTOSAR Interface is the connection to the RTE from both, the upper and the lower layer.

The VFB defines the communication pattern used in AUTOSAR. All communication is supposed to be asynchronous, which means that the sender of a message doesn’t block program execution even if the component expects a response. The VFB assigns a port to every component such that they can be uniquely identified. The VFB allows the client-server pattern for the communication of two components as well as a multicast sender-receiver-pattern enabling more than one component to receive a message.

Below the RTE the basic software consists of an operating system, services, a communication part, special device drivers, the ECU (Electronic Control Unit) abstraction, and the microcontroller abstraction. The operating system is based on OSEK OS. The services
provide functions like diagnostic protocol, NVRAM, flash and ram memory management. The communication part offers functions for the I/O and network management. The ECU and the microcontroller abstraction hide the real hardware from the software to build hardware independent software components. The complex device driver can add additional interfaces for microcontroller specific functionality. The ECU hardware is on the lowest level, which comprises sensors, actuators, and microcontrollers.

DESIGN FLOW

AUTOSAR does not only define an architecture and interfaces but also a design flow that specifies how software, given as software components (SW-C), should be mapped to the ECUs during the development cycle. Beside the interface definition of the SW-Cs a system configuration is required, which contains the descriptions of the ECUs and constraints that must be satisfied. During the design cycle the engineer maps the SW-Cs to the available ECUs such that the constraints in the system configuration are met. At the end an executable object of the software component is generated.

The idea of SW-Cs and the mapping to ECUs is a crucial point both in AUTOSAR and our proposed organic middleware. The difference is that AUTOSAR defines a static mapping at design time while the organic middleware maps the SW-Cs – called services in the middleware – to the nodes as required during the initial configuration and at runtime.

ORGANIC MIDDLEWARE FOR AUTOMOTIVE APPLICATIONS

The organic middleware for automotive applications is the outcome of a consequent feature improvement introduced by AUTOSAR. The architectural approach is based on the results of the investigations of the Organic Ubiquitous Middleware called AMUN or OCμ [6,7].

AUTOSAR introduces descriptions for the hardware as well as for the software components. The overall system is described with constraints that must be met. The descriptions of single ECUs can be extracted from the system description to find a suitable mapping of the SW-Cs to the ECUs at design time. We extend this mechanism to work also at runtime. It should be possible to transfer software components or services between the ECUs or nodes at runtime. This feature builds the basis for the self-configuration and self-healing mechanism of the organic middleware.

The Runtime Environment of AUTOSAR adds an abstraction layer to the middleware and defines the communication mechanism. Additionally the organic middleware assumes a common runtime environment on all nodes. This concept is known from Java with its virtual machine (VM) where a Java program can run on every computer that provides such a runtime environment. Another possible approach is the usage of so-called Universal Binaries (UB) known from Mac OS X where an executable comprises the binaries for PPC and Intel CPUs. If not too many different microcontrollers are used in one system the UB approach is preferable because the implementation of a VM needs much more effort and must be adapted to every new microcontroller. Nevertheless, both approaches allow the transfer of services from one node to another without the need for a recompilation.

A third requirement of the organic middleware is the need to split the black boxes into their basic parts like sensors, actuators and microcontrollers, which execute the services (software components). To gain the self-configuration and the self-healing capabilities an additional monitoring is needed to collect runtime information about the system and the services.

In the remainder of this section we explain the architecture of the organic middleware, give a brief overview of our configuration description, and introduce our metrics used to calculate the Quality of Service. Afterwards we describe the self-configuration and the self-healing mechanisms in detail.

ARCHITECTURE

The architecture of the organic middleware for automotive applications is the combination of the Organic Ubiquitous Middleware OCμ and the AUTOSAR architecture. The architecture of one node is shown in figure 2. The main parts are identical to the AUTOSAR architecture: the ECU as basis of the node, the microcontroller abstraction for the runtime environment, and the communication driver for the network communication. Every node is attached to the network via a bus. The bus can be one of the currently used busses like CAN, MOST, FlexRay, etc.

The major differences to AUTOSAR are the runtime environment, which is identical on every node in term of
the execution environment and the integration of the self-x services. Figure 3 shows a network with controller, sensor and actuator nodes, which are connected via a bus. The architecture is the same for all nodes of the network. They differ in the kind and amount of services and the computational power they offer. Controller nodes offer more CPU power to host services while sensors and actuators are limited to their special functions. This approach has the advantage that the same hardware components for the communication infrastructure can be used for all nodes. Thus they can be manufactured much cheaper due to huge amounts and the communication driver of the nodes must be implemented only once.

The communication pattern used in AUTOSAR is identical with the one used in the Organic Ubiquitous Middleware OCμ [6,7] except that the latter one offers more flexibility in the way messages can be delivered between the services.

CONFIGURATION DESCRIPTION

The design cycle of AUTOSAR defines some configuration files needed to map the SW-Cs to the ECUs. The organic middleware also defines two configuration files, one for the services of the applications and one for every node. The configuration file of a node describes the available resources and the attached sensors or actuators. The configuration file of the application describes the resource requirements of the services and dependencies, which describe the requirements of a service in form of other services, sensors, or actuators. The configuration descriptions are given in XML and can be easily extended to meet new requirements. A detailed description of the configuration description language and the XML-schema can be found in [11]. In the following an example for the definition of an actuator is shown.

```
<actuator id="windowlift_back_left" name="windowlift"
class="de.uau.sik.actuator.impl.WindowLift">
  <resource name="motor">
    <value name="speed" unit="rotation/s">5</value>
    <value name="lifting_force" unit="gramm">500</value>
  </resource>
</actuator>
```

A service and the possible dependencies can be defined as follows:

```
<service id="window_lifters" amount="1" name="WindowLiftSystem" priority="2"
class="de.uau.sik.service.impl.WindowLift">
  <resource name="cpu">
    <value name="frequency" unit="quantiSpeed">10</value>
  </resource>
  <resource name="ram">
    <value name="size" unit="kB">512</value>
  </resource>
  <resource name="programSize">
    <value name="size" unit="kB">100</value>
  </resource>
  <dependency type="actuator" amount="1" priority="1">windowlift_back_left</dependency>
  ...
  <dependency type="sensor" amount="4" priority="1">push_button</dependency>
  <dependency type="service" priority="1">aircondition</dependency>
</service>
```

METRICS

The self-configuration and the self-healing try to find a distribution of the services to the available nodes such that the overall resource utilization is as good as possible, i.e. that the resources are consumed equally on all nodes of the network. Therefore a metrics is defined to calculate a Quality of Service (QoS).
Based on the definitions in the configuration description every node can calculate the QoS of every service based on the locally available resources. We use a very simple metrics for the calculation of the QoS to show that it is possible to find good solutions with an intuitive approach. On the other hand we need only little computational power to calculate the QoS, which makes it more practical for small embedded systems with resource constrained devices like sensors or actuators. Furthermore the calculation of the QoS is a completely distributed mechanism without any central control. Figure 4 visualizes the possible situations that can occur in the calculation of the QoS.

Every resource can have multiple resource values. Each of them is defined by a maximum value $V_{\text{max}}$. The remaining (available) amount of the resource value is defined by $V_{\text{av}}$. The amount of a service’s required resource is defined by $V_{\text{req}}$. It is always true that $V_{\text{av}} \leq V_{\text{max}}$. Three cases can occur in the calculation of the QoS.

1. $0 < V_{\text{req}} \leq V_{\text{av}}$: In this case enough of the required resource is available to assign the service to the node.

2. $0 < V_{\text{av}} < V_{\text{req}}$: In this case the resource requirement exceeds the available resource, which might overload the node.

3. $V_{\text{av}} < 0 < V_{\text{req}}$: In this case the resource value of the node is already exhausted.

It depends on the resource if an overload is reasonable or not. For the CPU an overload might be acceptable while the overload of the memory might not be possible.

If a service is assigned to a node, the resources of the node $V_{\text{av}}$ are reduced by the amount of $V_{\text{req}}$. If $V_{\text{req}} > V_{\text{av}}$ the resulting value of $V_{\text{av}}$ is negative.

The second and the third case can be treated in the same manner, so the resulting resource consumption for one resource value $b_j$ is calculated by the following formula:

$$b_j = \begin{cases} 1 - \frac{V_{\text{av}} - V_{\text{req}}}{V_{\text{max}}} & \text{if } V_{\text{av}} \geq V_{\text{req}} \\ \frac{V_{\text{av}} - V_{\text{req}}}{V_{\text{max}}} & \text{otherwise} \end{cases}$$

The QoS for one resource can depend on more than one value that is defined for a resource. Therefore the Quality of Service $qos_i$ for the resource $i$ is calculated by the average of the sum of the resource’s values $b_j$. The average value is multiplied by a constant value $c$, normally 100, so the resource consumption can be expressed by a percentage value.

$$qos_i = c \frac{1}{m} \sum_{j=1}^{m} b_j$$

The overall QoS of a single service is calculated by the average of the sum of all resource consumptions.

$$qos = \frac{1}{n} \sum_{i=1}^{n} qos_i$$

The given QoS metrics tries to find the best match for all services, i.e. if a service completely consumes the remaining resources, the QoS would be the maximum of 100. The less the required resources match the available resources the worse is the QoS.

SELF-CONFIGURATION

The target of the self-configuration is to distribute the services given in the configuration description such that the resources of the nodes are consumed equally. This means that the load on the nodes generated by the services should be approximately the same.

The self-configuration is based on the social behavior of cooperative groups. For the explanation of the self-configuration we will concentrate on the basic mechanism and the parts needed to understand the evaluation results. The principal self-configuration approach is described in more detail in [8].

To distribute the services given by the configuration description the nodes of the network communicate to find the best assignment of the services to the nodes with respect to the available resources.

In the first step of the self-configuration the configuration description is given to one node, which floods the description to all other nodes of the network. Then every node calculates the QoS for every service based on the locally available resources and orders the services in descending order. Services that cannot be hosted due to missing resources are left out.
The second step of the self-configuration is the assignment phase. One of the nodes marks one of the services it wants to host and sends an assignment message to the other nodes, containing the id of the service and the QoS. The other nodes receive the message and mark the designated service as assigned to the node that sent the assignment message. They also save the QoS of the assignment, which will be needed if another node wants to overwrite an assignment. After a node has assigned a service it decreases the available resources by the amount of the resource consumption given in the configuration description and recalculates the QoS for the remaining services. The nodes continue to assign the services until all services of the configuration description are assigned.

If a node wants to host an already assigned service it sends an assignment message with the higher QoS. A node will only reassign a service if the QoS is at least 10% beyond the already given assignment. The nodes that receive the reassignment message just overwrite the assignment. The node whose assignment was overwritten can free the occupied resources.

After all services are assigned to the available nodes a verification step is employed to verify that every node has the same service assignments. Therefore one node sends its list with the assigned services and the QoS. If a receiver has a worse assignment for at least one node, it can overwrite the assignment with the better one given by the assignment list of the verification message. On the other hand, if a node has a better assignment for any of the services it sends a new verification message with the better assignments. If none of the nodes sends a new verification message within a predefined time, the verification step is assumed to be finished and the nodes can start the services.

During the assignment phase two nodes could send an assignment message at the same time for the same service leading to an inconsistent configuration. Conflicts can be avoided with four additional integer values added to each assignment message. The values are the amount of running services, the amount of already assigned services, the amount of services that can be still assigned to the node, and the id of the node. The four values of the conflicting nodes are compared against each other in four conflict resolution steps one after another. At the end, if all of the former values were equal, the id of the node is used to find a solution to the conflict because every node has a unique id. So a conflict can be solved locally on every node without the need for additional messages. The same applies to the other nodes that must also decide, which of the conflicting nodes will get the service assignment.

The presented self-configuration does not find an optimal solution for the distribution of the services. This is known to be an np-hard problem, which means that the effort needed to calculate the optimal solution is non-deterministic and needs a polynomial-time. Our approach was to find a simple and distributed mechanism that solves the problem with a minimum effort, knowing that the solution is only suboptimal, but good enough as starting point for a dynamic system where the real runtime behavior is hard to predict. The self-optimization mechanism presented in [9] can further optimize a system at runtime.

**SELF-HEALING**

As mentioned before failures occur even if the hardware and software were correctly designed and developed. Failures happen because of unexpected conditions and external changes, which could not be foreseen. Two kinds of failures can be distinguished: (1) software failures where parts of the software cannot compute correct values because the defined parameter range is exceeded or the software stopped running, and (2) hardware failures where a part of the hardware or a complete microcontroller is defect such that it can no longer be used to run services.

Currently both situations might bring an automobile to a full stop depending on the affected part. To avoid such situations an embedded system should be able to recover from failures. The proposed organic middleware and the software architecture based on services, where the services can be transferred to other nodes, build the basis for the self-healing mechanism.

The target of the self-healing is to ensure that the needed services, as described in the configuration description, are available at any time. This enfolds the detection of failures as well as the healing. We also investigate a detection mechanism that is described in detail in [10]. In this paper we focus on the healing part, which presupposes that a failure has already been detected.

The self-healing is based on the configuration mechanism of the self-configuration. If a node or a service crashed, the self-healing tries to rebuild a valid configuration with the left hardware and resources. A partial configuration description is created out of the original configuration description containing the services that are affected by the failure. The node that detected the failure floods the partial configuration description into the network and the nodes run the same algorithm as for the self-configuration. The difference is that the nodes are already loaded with some services. At the end of the self-healing a verification step assures that all nodes have the same assignment of the services.

It might happen, that the required resources for the services that are subject to the self-configuration are not available. This might lead to an unsatisfiable configuration, i.e. at least one of the services does not get the required resources. It depends on the resources if an overload, which is expressed by a negative QoS, can be tolerated or not. Especially small networks might be short of redundant resources. In the evaluation section we give an example of a small network where unsatisfiable configurations occur more frequently during...
a self-healing than in larger networks, even if the overall load is the same in both networks.

EVALUATION

To evaluate our organic middleware we implemented a simulator with an abstract and simplified view on the architecture. The main interest was on the quality of the self-configuration and the efficiency of the self-healing. The first part of the evaluations is on the self-configuration and the second on the self-healing.

Every evaluation was run 100 times. The simulations were run with network sizes of 10, 25, 50, and 100 nodes. We will concentrate on the networks with 25 and 50 nodes, which covers the most common setups. The nodes are generated at the beginning of the simulations with 60% homogeneous and 40% heterogeneous nodes. The homogeneous nodes are created with the same amount of resources. For the heterogeneous nodes the resources are varied to create a network with different types of controller nodes.

The services of the configuration description are generated such that the nodes are loaded to 60%. We also chose a mixture of homogeneous and heterogeneous services. This means, that a given amount of services have the same resource requirements while the resource consumptions of the remaining services are varied.

For the conclusion about the efficiency of the algorithms the amount of messages and the reached QoS are of interest. An optimum can be defined for the amount of messages needed to complete the self-configuration process. One message is needed to flood the configuration description, \( n \) messages must be exchanged to assign the services to the nodes, and one message is required for the verification step. Therefore the lower bound for the messages is \( n + 2 \).

A suboptimal case can be defined to see how many reassignments and verification messages are required. Assuming that during the self-configuration every node tries to overwrite one assignment or sends one additional verification message, the suboptimal case needs \( n + m + 1 \) messages, where \( m \) is the amount of nodes in the network and \( n \) the amount of services in the configuration description, which must be assigned to the nodes. The optimal and the suboptimal case are shown as two lines in all the charts showing the amount of messages as reference points.

In the next section the evaluation results of the self-configuration are shown followed by the self-healing results. In the self-healing section first the results for node failures, which might affect more than one service, and then the results for single service failures are presented.

SELF-CONFIGURATION

The self-configuration is used to assign the service of the configuration description to the nodes of the network depending on the available resources. The target is to find a distribution such that the resources of all nodes are consumed equally.

The interesting measures of the evaluations are the amount of messages needed to accomplish the self-configuration and the reached Quality of Service. The left chart of figure 5 shows the amount of messages needed for the self-configuration in a network with 25 nodes and 100 services. The two lines in the diagram are the optimal (lower line) and the suboptimal (upper line) case as described in the former section.

The bottom line shows the amount of verification messages during the verification step. The line with only a few peaks between the optimal and the suboptimal cases shows the amount of assignment messages not considering reassignments. This line is slightly above the optimum because some nodes send their assignment message at the same time. This leads to a conflict, which can be solved locally on every node. The line with small and huge peaks shows the total amount of message used for the self-configuration. For all simulations the total amount of messages is always between the optimal and the suboptimal case but most of the time it is much nearer to the optimal than to the suboptimal case. The dashed line shows the average amount of messages.

The Quality of Service of the simulations is shown in the right chart of figure 5. The values show the deviation of the QoS value from the expected value. The services are generated such that the nodes should be loaded to 60%. The deviation of the QoS is 0 if this value is exactly reached. The chart shows, that the average QoS is 0, which means that the average load of the nodes in the network is 60%. Additionally the deviation of the best and the worst QoS of each simulation run are given.

Figure 6 and 7 show the simulation results of a network with 50 nodes and 100 services and a network with 50 nodes and 200 services. The self-configuration performs well for all simulation setups and the amount of messages is always in the range of the optimal and suboptimal case. For the majority of the simulations the amount of messages is nearer to the optimal than to the suboptimal case. The average QoS is around 0 for all simulations, which confirms that the nodes are loaded to 60%. Also the deviation of the worst and the best QoS is in a range where the nodes are not overloaded.

The simulation results of the self-configuration show that it perfectly fulfills the posed task. The amount of messages is only slightly higher than the optimal case. The average amount of messages per service is 1.1. Furthermore the Quality of Service reaches the expected value, which results in an equal load for the nodes of the network.
Figure 5: Evaluation results with 25 nodes and 100 services

Figure 6: Evaluation results with 50 nodes and 100 services

Figure 7: Evaluation results with 50 nodes and 200 services
SELF-HEALING

The self-healing should assure that all the required services of the configuration description are available at any time. For the simulations we assume that the nodes of the network can detect a failure. The nodes try to recover from failures by reconfiguring the missing services as they did for the self-configuration.

The simulations for the self-healing comprise two scenarios: (1) node failures where all the services on a node become unavailable and (2) failures of a single service.

Node Failure

For every simulation run up to 40% of the nodes fail, but the nodes are not totally shut off. After a failure a node is expected to rejoin the network, but without any of the former services.

In small networks chances are high that a node with a unique resource fails and that no other node has a similar resource. In this case the self-healing cannot reestablish the system integrity. Figure 8 shows the simulation results for a network with 10 nodes. Some simulation runs show a cross, which means that the self-healing wasn’t able to recover from the failure due to missing resources. This situation is also detected by the self-healing mechanism. For some simulations the amount of messages is below the optimal case. This happens if a node crashed, which had no service. Thus no message for the self-healing was needed.

Figure 8: Self-healing in a network with 10 nodes

The former simulation demonstrates that the self-healing is also able to detect situations where the reconfiguration does not lead to the desired result.

More realistic simulation setups are shown in figure 8, 9, and 10. The charts show the simulation results for networks with 25 and 50 nodes as well as 100 and 200 services. It can be observed that the amount of messages is nearly at the optimal case and that only a few reassignment messages are sent during the self-healing.

Figure 9: Self-healing after node failures in a network with 25 nodes

100 services

Figure 10: Self-healing after node failures in a network with 50 nodes

100 services

Figure 11: Self-healing after node failures in a network with 50 nodes

200 services

Service Failure

In case of a service failure only one service crashes. The node, on which the service is running, is otherwise not affected by the service’s failure. Furthermore, 40% of all services are expected to fail in the following simulations.

Figure 12, 13, and 14 show the simulation results of networks with 25 and 50 nodes and 100 and 200 services. It can be observed that more messages are needed than in the former case of the node failures. This happens, because more than one node tries to get the service nearly at the same time. The nodes send an assignment message within a short interval, which leads to conflicts. The conflicts are solved locally so no additional message is needed. The charts also show that more reassignments are done than in the case of node crashes.

The most outstanding advantage of the self-healing is that the algorithm linearly scales with both, the network size and the amount of services in the configuration description. Other approaches show an exponential increase. It must be pointed out again, that 40% of the nodes fail during every simulation run.
Higher Failure Rate

In the former simulations 40% of the services were expected to crash. This is already a huge amount of services but the following simulations were done with a failure rate of 60%. This means, that 60% of the services crash during every simulation run.

Figure 15 and 16 show the simulation results for networks with 25 and 50 nodes each with 100 services. The amount of messages for the self-healing doesn’t increase significantly. Due to the higher amount of crashed services the total amount of messages increases but again the total amount of messages is in the lower part of the range from the optimal to the suboptimal case.

The average amount of messages per service for the self-healing ranges from 1.3 to 1.7 depending on the size of the network and the amount of services.

The self-healing needs more messages for the assignment of the failed services than the self-configuration for the initial assignment of the services because many nodes try to get one of the failed services. No mechanism is currently implemented to prevent the nodes from sending assignment messages at the same time. So if a node would be able to host a formerly crashed service it will send an assignment message during the self-healing.

We currently investigate a mechanism to prevent the nodes from sending assignment messages all at once. The idea is to delay the assignment messages depending on the calculated Quality of Service. The lower the QoS the later an assignment message will be sent. In the meanwhile another node with a higher QoS might already have sent an assignment message, which might obsolete the own message. The same mechanism might also help to further reduce the amount of assignment messages during the self-configuration.

RELATED WORK

The cooperative character of the nodes during the self-configuration process has strong similarities with Multi-Agent-Systems (MAS) [12]. The IEEE foundation FIPA [13] promotes standards for agent-based technology to foster the interoperability with other technologies. The Agent Technology Roadmap [14] also describes the usage of agents as a possible way to solve the complexity problems of self-* and autonomic computing systems.

An approach similar to the self-configuration is used to solve the problem of task allocation within a MAS, which is described by the Contract Net Protocol [15]. The agents bid for an announced task and the best agent is
needed for a self-healing is slightly higher than for the used for the self-configuration. The amount of messages same configuration capability of the organic middleware node and service failures. The self-healing employs the self-healing is able to recover the system from both, service. This value is only slightly above the optimal process. The amount of messages is 1.1 messages per configuration description during the self-configuration networks, which can respond dynamically to changes in the configuration policies. It has the drawback that it is a server-based approach with a single point of failure.

Regarding the self-healing only a few publications exist that describe a real system implementation. Some theoretical considerations where given by Koopman [17]. He describes the possible problem space for self-healing. The problem about self-healing is that it is very application specific and addicted to the kind of regarded failures. A lot of work is known from the area of fault tolerance but the most common way is to work with so called hot or cold spares.

CONCLUSION AND FUTURE WORK

In this paper an organic middleware based on the AUTOSAR architecture is presented, which exposes self-configuration and self-healing capabilities. The required changes for the AUTOSAR architecture to become an organic computing architecture are described. The major difference is to overcome the current black box design. Sensors and actuators are separated from the computational parts and connected via a network. Furthermore the applications should be implemented as a collection of services instead of a monolithic block.

The self-configuration and the self-healing mechanisms are described and simulation results are presented. The nodes of the network perfectly distribute the services of the configuration description during the self-configuration process. The amount of messages is 1.1 messages per service. This value is only slightly above the optimal case.

The self-healing is able to recover the system from both, node and service failures. The self-healing employs the same configuration capability of the organic middleware used for the self-configuration. The amount of messages needed for a self-healing is slightly higher than for the self-configuration but still inside the range of the optimal and suboptimal case.

The main advantage of the self-configuration and self-healing is the distributed nature of the mechanisms. No central control instance is required and only local knowledge is used. Both mechanisms can be scaled perfectly and show only a linear increase of messages with an increasing network size and amount of services. This means that the average amount of messages per service stay at the same level independent of the network size.

Future work will be to investigate strategies to further reduce the amount of messages especially for the self-healing. Moreover we are interested in measuring the speed of the self-configuration and self-healing to give an estimation how long a self-configuration or self-healing might take in a real environment. Therefore we need to extend the simulator not only to handle abstract simulation steps but also real time requirements. Furthermore we will build a real network with embedded controllers to extend the current AUTOSAR implementation to do some evaluations in a real setup.

REFERENCES

11. M. Helbig: Selbstkonfiguration in AUTOSAR, Universität Augsburg, Diplomarbeit, August 2006

CONTACT

Dr. Wolfgang Trumler
University of Augsburg
Eichleitnerstr. 30
86159 Augsburg, Germany
Tel.: (+49) 821 598 2355
Fax.: (+49) 821 598 2359
Email: Wolfgang.Trumler@informatik.uni-augsburg.de

Wolfgang Trumler studied computer science at the University of Karlsruhe, Germany, from 1989 to 1996. After six years of professional experience, two years as hardware engineer and for four years as CTO of the company Intermediate GmbH &Co. at Karlsruhe, he continued his scientific career at the Chair of Systems and Networking at the University of Augsburg as a scientific assistant and as a postgraduate researcher after finishing his Doctoral degree in July 2006.