Designing Self-healing in Automotive Systems

Hella Seebach[1], Florian Nafz[1], Jörg Holtmann[2], Jan Meyer[2], Matthias Tichy[2], Wolfgang Reif[1], and Wilhelm Schäfer[2]

[1] Department of Software Engineering and Programming Languages, University of Augsburg, 86135 Augsburg, Germany
{seebach,nafz,reif}@informatik.uni-augsburg.de
[2] Software Engineering Group, University of Paderborn, Paderborn, Germany
{jholtmann,jmeyer,mtichy}@s-lab.upb.de, wilhelm@upb.de

Abstract. Self-healing promises to improve the dependability of systems. In particular safety-critical systems like automotive systems are well suited application, since safe operation is required in these systems even in case of failures. Prerequisite for the improved dependability is the correct realization of the self-healing techniques. Consequently, self-healing activities should be rigorously specified and appropriately integrated with the rest of the system. In this paper, we present an approach for designing self-healing mechanisms in automotive systems. The approach contains a construction model which consist of a structural description as well as an extensive set of constraints. The constraints specify a correct system structure and are also used in the self-healing activities. We exemplify the self-healing approach using the adaptive cruise control system of modern cars.

Keywords: Organic Computing, Automotive Systems, Self-Organization.

1 Introduction

Self-x operations are increasingly utilized in today's systems in order to satisfy the functional requirements even in case of failures, unexpected environmental changes, etc. without any manual interventions. Self-healing deals with the detection and correction of partial system failures. Building a self-healing system is very challenging as it must address most of the possible failure scenarios and has to handle all repairable ones. Consequently, the self-healing part of the system should not be build in an ad-hoc fashion but rather be specified in a formal way based on proven platforms and best practices.

In previous works, a design pattern (Organic Design Pattern (ODP)) for self-organizing resource-flow systems has been developed [11,14,16] and evaluated in the context of production automation systems. In this work, self-organization is the basis for self-healing and other self-x properties. The design pattern formally specifies all relevant parts of the system and defines constraints, which have to be maintained in operable system states. Thus, a corridor (set of valid systems states) is defined. Whenever the constraints are violated, the system reconfigures

B. Xie et al. (Eds.): ATC 2010, LNCS 6407, pp. 47–61, 2010.

to return into this corridor to repair the system. This is called the Restore Invariant Approach.

Self-healing is also applicable to other domains. This especially holds for safety-relevant systems, for example avionics and automotive systems, which must operate continuously. In avionics, redundancy is the standard for fly-by-wire systems [8] to ensure a continuous and thus safe operation. However, redundancy is typically not employed for automotive systems due to the prohibitive costs and the tight integration of hardware and software which are typically sold as a single product by the automotive suppliers. Due to the increasing amount of software in automotive systems [10] and the resulting reliance on the safe operation of the software and electronics as well as the adoption of software standards like AUTOSAR [1], the usage of self-healing in automotive systems becomes feasible. There are already some projects [21], [2] working on the requirements needed for self-healing in the automotive domain. They establish concepts for the possibility of redundancy and how software components can be relocated on runtime. But these approaches do not specify how correct system states can be defined and thus, how guarantees about a correct system behavior can be given which is absolutely required for self-healing safety-relevant systems.

Automotive systems consist of a number of software components that are deployed on independent computing units in the car. Data is exchanged between these components over different buses, examples of data are sensor data, status reports, or commands. In order to describe the valid system states of such "data-flow systems", we adapted the ODP and the restore invariant approach. The techniques were then applied to the adaptive cruise control system of a car.

In the next section, we present the adaptive cruise control system which is used as a running example. Section 3 contains a presentation of the ODP as well as its adaptation to data-flow systems including the constraints defined for this domain. Thereafter, we present the application of the adapted ODP to the adaptive cruise control including self-healing scenarios in Section 4. After a discussion of related work in Section 5, we close with a conclusion and an outlook on future work.

2 Adaptive Cruise Control

In a modern car more and more functions are realized with software [9]. This includes safety-relevant functions like driver assistance or in the future X-by-wire systems. These functions need special techniques like redundancy to guarantee correct behavior. Examples for safety-relevant systems are advanced driver assistant systems. These systems assist the driver to avoid accidents. They use sensors to identify dangerous situations. If such a situation is detected the driver is warned visually or auditory. It is even possible that the assistance systems can regulate the car, to prevent dangerous situations. Therefore, the system has to interact with other subsystems in the car.

As a typical example for a safety-relevant driver assistance system, we use an adaptive cruise control (ACC) in this paper [22]. It is an extended speed

control. The functionality is that if no obstacle is detected it accelerates the car to the speed which was entered by the driver. If an obstacle is detected the car is decelerated, so that there is an adequate gap between the car and the obstacle (mainly another car).

The structure of the ACC system is presented in Fig. 1. The figure shows the system structure by means of electronic control units (ECUs) with software components (...-SW) deployed to it. The ECUs are connected to each other by bus systems enabling message exchange. Different bus systems have to be connected by a gateway, which translates between the different bus protocols and handles throughput differences.

The ACC system normally consists of a speed sensor, an object detection system, and a control unit. The object detection system is either a radar (radio detection and ranging) or lidar (light detection and ranging). Using either device, the ACC detects if there are obstacles in front of the car. If one is detected, the car's speed is adapted. This is first done with the engine brake. If this is not enough then the brakes are activated, too. Thus, the ACC is in contact with the *BrakeControl-ECU* and the *EngineControl-ECU* and additionally with the *LightControl-ECU* to show the braking situation to other traffic participants.

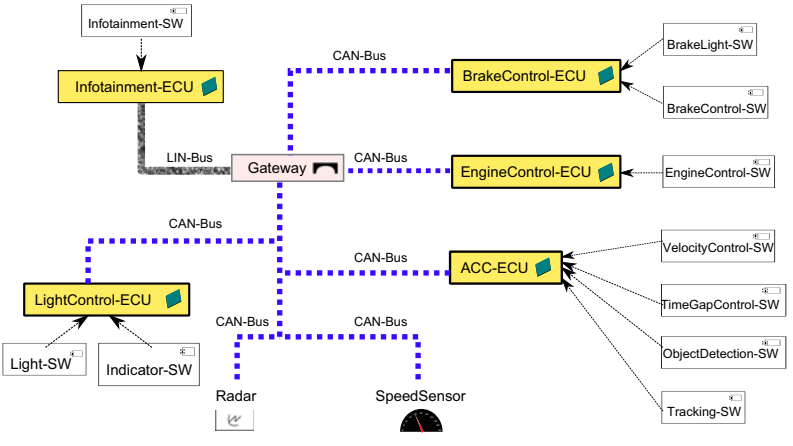

Fig. 1. Overview of the ACC system structure

In this example we use some abstractions for the ACC system. We assume that every control unit is connected to a communication bus. Furthermore, the sensors are directly connected to the bus, so they are called intelligent sensors in the automotive domain. We also assume that it is possible to reconfigure the software at runtime. This is currently not possible since the automotive manufacturers configure the ECUs' functions statically at design time. But in the future online reconfiguration will become more and more interesting and will be realized. During the reconfiguration process the ACC system is deactivated and then restarted. This is currently a common procedure to handle software

errors. When the system is deactivated the driver is informed that he has to drive manually without any help. But in the future it is a possible scenario that redundancy enables the continuous operation even during the reconfiguration process. The AUTOSAR standard [1] with its standardized interfaces and its run-time environment (RTE) is the first step towards a system that can be reconfigured.

We have chosen an example from the automotive industry because it is a typical representative of a data-flow system. The characteristics of such systems are that data are produced (e.g., by a sensor) and sent to other systems in a next step. These can be actuators or other software subsystems. There, the incoming data are used to execute an action or to calculate new commands. Thus, in data-flow systems the data are sent from one system to another, so data chains are established. This is very similar to resource-flow systems where resources are sent from one agent to another but there are some differences like the possibility to use data parallel on different agents. This is formulated in the next section.

3 The Organic Design Pattern for Data-Flow Systems

The organic design pattern (ODP) has been developed to design and construct self-organizing systems in a top-down manner. This section describes shortly the ODP for resource-flow systems (RFS) and then maps to data-flow systems like automotive systems. In particular the constraints required to describe this domain are discussed.

3.1 ODP for Resource-Flow Systems

As the complexity of modern software systems increases steadily, the administration and maintenance becomes more and more time intensive. Therefore the ODP has been defined to develop such systems in a top-down manner and to specify guarantees for these systems which are maintained by the system itself without external control. Previously, the ODP was defined for self-organizing resource-flow systems. One major contribution of this paper is the adaptation of the ODP-RFS to data-flow systems. But first the ODP-RFS and its concepts, goals, and mechanisms are described.

The ODP-RFS combines a top-down engineering approach [17] with self-organizing concepts which allow, for example, self-healing behavior. The pattern defines the main concepts of resource-flow systems with self-x behavior and constraints which are modeled to define correct system states. Based on these concepts the respective engineer is able to define additional domain specific constraints. Fig. 2 shows the main concepts the ODP-RFS also includes, so for a better understanding of the following explanations please refer to this figure.

Self-healing requires some kind of redundancy in the system. In systems designed with the help of the ODP-RFS the redundancy is achieved by redundant *capabilities* the system components called *agents* have. Additional redundancy comes into the system by variable resource-flow possibilities or communication

possibilities respectively. The agents have *roles* specifying from which agent they receive *resources*, which capabilities to apply to the resource and to which agent then to hand over the processed resource.

The agent is restricted in its behavior by constraints and is able to permanently monitor these constraints. In case of a violation of at least one of the constraints, the agent needs to heal the system. This means the agent starts a reconfiguration process which in the context of the ODP-RFS is a reallocation of roles to the agents. The constraints specified on system class level define what a correct role allocation looks like and thus define a correct resource-flow within the system. One aspect of correctness here means that every resource entering the system leaves the system processed by all capabilities the resource needs (specified in the *task*). The role allocation can be calculated by several mechanisms, for example a constraint solver or distributed algorithms, tailored to restore the constraints. More details to the ODP-RFS can be found in [11].

How the concepts of the ODP-RFS and the constraints mentioned can be used in the class of data-flow systems, especially automotive systems, is described in the following section.

3.2 Adapting the ODP to Data-Flow Systems

Considering data as resources, data-flow systems (DFS) as they occur for example in automotive systems are in several points very similar to resource-flow systems. They only pass data instead of resources to other *agents*. Therefore the ODP-DFS inherits many concepts and constraints already defined in the ODP-RFS. Fig. 2 shows the adapted ODP. Similar to resource-flow systems, data-flow systems consist of agents. In the ODP-RFS the *agents* are, for example robots or autonomous vehicles while in ODP-DFS the *agents* are, for instance ECUs. Each *agent* has different *capabilities* which are used to work on the data or change the state of the data. In the ACC example data from the radar sensor is sent to different software components (represented by the capabilities) like *Tracking-SW* or the *ObjectDetection-SW* which calculates data dependent on the radar data.

In data-flow systems the *agents* have different types of properties they provide, for example a certain amount of memory or the clock speed of an ECU. The different properties are necessary to allow parallel processing of data, a common feature in data-flow systems. Accordingly the *capabilities*—mainly software components—an *agent* has, require several types of properties of the agent. This fact is encapsulated in three new concepts, the type of the property (*PropertyType*), the required property (*RequiredProperty*) and the provided property (*ProvidedProperty*). The properties are used to describe which capabilities can be allocated to which agent (e.g., which software can be deployed to which ECU).

In RFS the *task* has been considered as an ordered sequence of *capabilities*. For a consistent terminology the term is also used in the ODP-DFS. But the term task should not be mixed with the term task that is used in the automotive industry. There the term is used for an operating system resource. Here it is used for the specification of capabilities. In data-flow systems the *task* is more complex than in resource-flow systems because the data can be used in different

software components simultaneously. For example the radar sensor data are used in the *Tracking-SW* as well as in the *ObjectDetection-SW*. As a new functionality of the ODP-DFS the data could easily be split up or merged. For that purpose every *capability* defines its *posts*, that means which *capabilities* follow in the processing order. Thus, the *task* is a graph of *capabilities* with a partial order.

Another new concept in the ODP-DFS is the *channel* concept. In resource-flow systems the *agents* are able to hand over their resources directly to other *agents*. In data-flow systems the agents are connected via communication buses like CAN or LIN which are used for example in the automotive industry. The *channel* in the ODP-DFS models these communication buses. An *agent* is able to communicate to all other *agents* connected to the same *channel*.

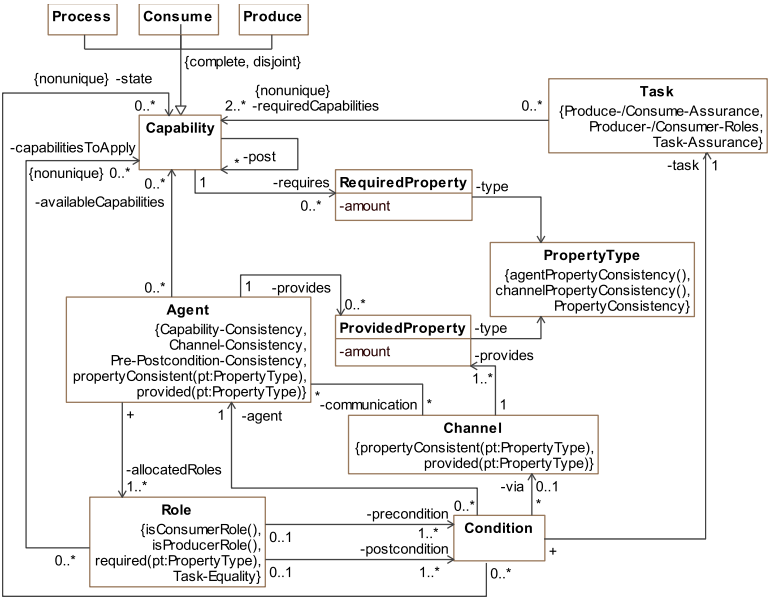

Fig. 2. Organic Design Pattern for Data-Flow Systems

The concept of *roles* in data-flow systems is very similar to the concept in the ODP-RFS. The *role* determines which *capabilities* an *agent* performs and from which *agent via* which *channel* it receives the data or to which *agent via* which *channel* the *agent* has to send the data. The difference here is only the *via* concept of the *pre- and postconditions* which is in addition to the *role* concept from the ODP-RFS.

In some of the classes of the ODP-DFS OCL-constraints[1] are defined to specify correct system states. For example in the class task, there are three constraints defining a correct task or role allocation to agents (*Producer-/Consumer-Assurance, Producer-/Consumer-Roles, Task-Assurance*). In the next section the

[1] Object Constraint Language, OMG Available Specification.

constraints are explained in detail and are represented in a formal notation (standard logic notation). In [7,14] the translation from OCL-constraints to a standard logic notation is presented.

3.3 Constraints

As mentioned above, a correct configuration of the system (which leads to correct behavior) is defined by constraining the system. This is done by annotation of OCL-constraints to the ODP-DFS. In case of a failure, one or more of the constraints are violated and the system needs to calculate a new configuration, such that the constraints hold again. This can be done by using a standard off-the-shelf constraint solver, like KodKod [19], Alloy [12], or Cassowary [5]. In [14] this specification approach with constraints is presented for ODP-RFS systems and non-numerical constraints. As described in Section 3.2, the major extensions besides some artifacts like the channel concept are much more complex tasks in form of graphs as well as quantitative restrictions, for example for bus load or RAM. In the following we describe some of the constraints for the ODP-DFS, especially the more complex ones resulting from the extensions of the ODP. For better readability constraints are written in standard logic notation not in OCL notation.

Basic Constraints: These constraints are a standard set for the role model proposed with the ODP. They just need to be adopted to the designation used in ODP-DFS and the additional associations. The values which can be assigned in a role must adhere to the agents situation. For example, only available capabilities can be assigned or channels which the agent is connected to. This is expressed in the two constraints *Capability-Consistency* and *Channel-Consistency*.

Capability-Consistency:

$$\forall a \in Agent\ , \forall r \in a.allocatedRoles : \\ r.capabilitiesToApply \subseteq a.availableCapabilities$$

Channel-Consistency:

$$\forall a \in Agent\ , \forall r \in a.allocatedRoles : \\ (\forall c \in r.precondition : c.via \subseteq a.communication) \\ \wedge (\forall c \in r.postcondition : c.via \subseteq a.communication)$$

A more complex constraint is ensuring that if one agent has a role which tells it to send data to another one via a specific channel, the other one must have a role assigned, with a precondition, telling him that he receives data via this channel. So they must be *connected* with respect to their role assignment.

Pre-Postcondition-Consistency:

$$\forall a \in Agent\ , \forall r \in a.allocatedRoles\ : \\ (\forall c \in r.postcondition\ , \exists a_{rec} \in c.agent : connected(a, a_{rec})) \\ \wedge (\forall c \in r.precondition\ , \exists a_{send} \in c.agent : connected(a_{send}, a))$$

$$connected(a_{send}, a_{rec}) :\Leftrightarrow \\ \exists r_{send} \in a_{send}.allocatedRoles\ , \exists r_{rec} \in a_{rec}.allocatedRoles, \\ \exists c_{send} \in r_{send}.postcondition\ , \exists c_{rec} \in r_{rec}.precondition :$$

$$c_{send}.agent = a_{rec} \wedge c_{rec}.agent = a_{send}$$
$$\wedge c_{send}.via = c_{rec}.via \wedge c_{send}.task = c_{rec}.task$$

Further basic constraints are for instance, that a producer role has an empty set of preconditions (*isProducer*) and analogously a consumer role an empty set of postconditions (*isConsumer*). *Produce-/Consume-Assurance* states that roles with produce or consume capabilities must apply the first or the last element of the task. Likewise, the first and the last element of a task has to be a produce and a consume capability, respectively (*Produce-/Consume-Assurance*), see for example Fig. 4.

Extension to graph like tasks: Further it must be assured that the state within a role always conforms to the actual task.

$$\forall c \in Condition : c.state \sqsubseteq c.task$$

Here "$\sqsubseteq$" formally denotes a prefix relationship between state and task. In [14] we just considered simple tasks in form of lists, with strongly sequential application of capabilities and no forks. Therefore the prefix relationship for lists was sufficient. In the domain of data-flow-systems we now have tasks which are graph like structures, describing the dependencies on the processing steps. Several successors mean that the result needs to be processed by each one and therefore for example transmitted to several agents which have the required capabilities assigned. For a task T described as a graph a state S is a subgraph describing the part which is already done. The prefix property over graphs is defined as follows:

Task-Assurance:

$$\begin{aligned} S \sqsubseteq T :\Leftrightarrow\ & V(S) \subseteq V(T) \\ & \wedge E(S) \subseteq E(T) \\ & \wedge S = T - (V(T) \backslash V(\mathcal{C}_{T^{-1}}(V(S)))) \end{aligned}$$

where $T - (V(T) \backslash V(\mathcal{C}_{T^{-1}}(V(S)), V(S))))$ is the subgraph induced by building the transitive closure over T^{-1} (T with reverse edge relation) starting from the nodes in S.

Further we need a constraint describing the effect of the application of a capability. The state in the postcondition must be the result of the application of a capability to the state in the precondition. We formally express that with the function "++".

$$\forall r \in Role : \forall\ c_{post} \in r.postcondition :$$
$$c_{post}.state = r.precondition.state + + r.capabilityToApply$$

For lists and one precondition, ++ is the standard list concatenation operator. Extending the definition to graph like structures and multiple conditions in roles leads to the following definition:

$$S + + c = \left(\bigcup_{s \in S} s \right) \cup \left(\bigcup_{v \in fin(S)} (v, c) \right)$$

where $fin(S)$ is the set of all sinks of all state graphs in S and S is a set of state graphs and c a capability.

Extension for parallel role execution: During reconfiguration a simple assignment of the capabilities to one agent that can perform them and ensuring that the data can be sent via a channel to the next one is not sufficient. Usually, capabilities require some amount of properties, like memory or processor power, which the agent needs to provide. As the roles are executed in parallel, an agent might not be able to perform a capability because it lacks memory as it is reserved to several other assigned capabilities. The following constraint makes a pessimistic assumption to assure that capabilities are only assigned if there is enough amount of needed properties available. Same holds for the channel, as it is not exclusive, it needs to be ensured that the load of the channels is not greater then the bandwidth provided[2].:

$$
\begin{aligned}
&Property\text{-}Consistency:\\
&\quad \forall pt \in PropertyType :\\
&\quad (\quad (\forall a \in Agent :\\
&\qquad \Big(\textstyle\sum_{r \in a.aRole} \sum_{c \in r.capToApp} c.requires(pt)\Big) \leq a.provides(pt))\\
&\quad \vee\ (\forall ch \in Channel :\\
&\qquad \Big(\textstyle\sum_{r \in aRole|_{r.pc.via == ch}} \sum_{c \in r.capToApp} c.requires(pt)\Big) \leq ch.provides(pt)))
\end{aligned}
$$

There are a few further constraints derived from the cardinalities of the pattern. A cardinality of one leads to a constraint that there must be exactly one element. For example,

$$\forall c \in Condition : \|c.task\| = 1$$

states that there is exactly one task connected to a condition.

Together these constraints express a valid configuration of the system (a valid role allocation) and whenever a failure occurs the system needs to reconfigure according to these constraints.

4 Application of the ODP to the Automotive Domain (ACC)

In Section 3 the ODP and the specific ODP-DFS for data-flow systems have been described. In this section a specific automotive system, the ACC, is modeled with the ODP-DFS. This is done in a simplified way while abstracting from technical issues. Thus, the functionality of the ODP-DFS is shown and can be understood without knowledge how the ACC system works in detail. The instantiation of the ODP-DFS yields to several diagrams which specify the structure of the ACC example. These diagrams are called domain models.

[2] aRole := allocatedRoles, capToApp := capabilityToApply, pc := postcondition.

Fig. 3 shows the domain model of a *16bitAgent* representing a 16-bit ECU. In the ACC scenario four types of agents exist: A 16-bit agent, a 32-bit agent, a gateway agent, and a sensor agent. These agent types represent ECU types (ECUs with word size of 16/32 bit) or other types of hardware devices (gateway/sensor). A domain model exists for every agent type which determines what kind of capabilities the agent type has in general (e.g., which software components can be deployed to a 16-bit ECU) and which properties the agent provides. Additionally the properties required by the capabilities are defined. In Fig. 3 the small annotation in the upper right corner indicates which general ODP-DFS concept the domain specific concept is derived from.

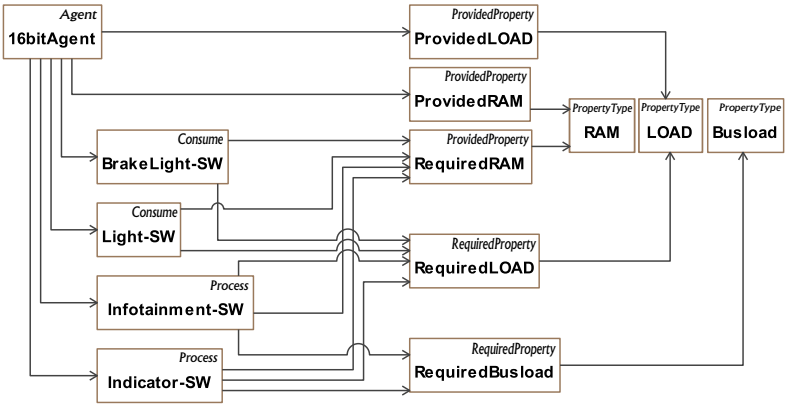

Fig. 3. Domain model for a 16-bitAgent

A *16bitAgent* has four capabilities, *BrakeLight-SW*, *Light-SW*, *Infotainment-SW*, and *Indicator-SW*. All these capabilities represent software components (SW) which can be deployed on any ECU with a 16-bit word size. For example in Fig. 1 the components *Light-SW* and *Indicator-SW* are running on *LightControl-ECU*. These components could also run on a different 16-bit ECU. In contrast, 32-bit ECUs can compute bigger values and can address more memory than a 16-bit ECU. Thus, they are able to deal with more time and resource consuming operations needed by software components like *ObjectDetection-SW*, *VelocityControl-SW*, *Tracking-SW*, *TimeGapControl-SW*, *EngineControl- SW*, and *BrakeControl-SW*. A 32-bit ECU can also handle software components which are designed for 16-bit ECUs.

The agent (ECU) provides *RAM* and a certain *LOAD* which is required by the capabilities running on the agent. Every capability (software component) that runs on it, decreases the provided load by a certain value. So the capabilities are categorized as *produce*, *consume* and *process* capabilities as shown in Fig. 2. This means that some capabilities produce (e.g., sensors) bus load if they send the data via channels to another agent or just process (*Indicator-SW*) the data (also producing bus load), while other capabilities consume the data (*BrakeLight-SW*)

and thus do not require any bus load. The need for free bus load of the channel is encapsulated in the concept of the *RequiredBusload* which must accordingly be provided by the bus the 16-bit ECU is connected to. In Fig. 3 the missing links between *BrakeLight-SW/Light-SW* and *RequiredBusload* also indicate that these capabilities do not produce any bus load and thus are consuming capabilities.

After defining the agents in the scenario with all its capabilities and properties the next step is to define the task the system has to fulfill. As already mentioned, a task is a graph of capabilities with a partial order. Fig. 4 shows the task for our running example ACC. Every depicted capability fulfills a part of the ACC functionality and represents the corresponding software component in Fig. 1.

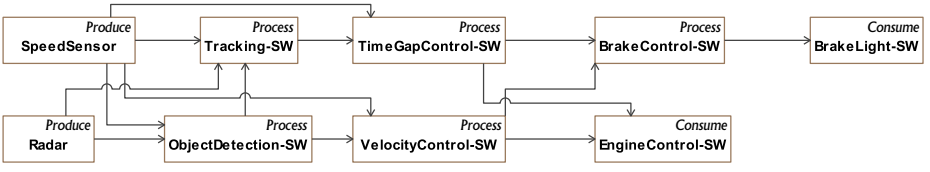

Fig. 4. A task for the ACC example

The task starts with the data producing capabilities *Radar* and *SpeedSensor*, which deliver sensor data to the dependent capabilities *ObjectDetection-SW* and *Tracking-SW*. *ObjectDetection-SW* keeps track of the current velocity and of objects within the driving direction of the vehicle. Based on this data and the stored desired speed, the software component decides whether the ACC has to apply velocity or time gap control. In velocity control mode the component *VelocityControl-SW* is active. In this case it adjusts the current velocity to the desired velocity by sending signals to *EngineControl-SW* and *BrakeControl-SW*. In time gap control mode, the components *Tracking-SW* and *TimeGapControl-SW* are active. The first one determines and filters relevant target objects from the complex radar data. Afterwards, *TimeGapControl-SW* compares the vehicle's current velocity with the velocity of the relevant target objects and operates engine and brakes via *EngineControl-SW* and *BrakeControl-SW*, respectively. When the brake is actuated, the brake lights are turned on by sending a signal to *BrakeLight-SW*.

When all agents and the task for the system are defined, it is possible to specify scenarios for self-healing. This is presented in the next paragraph.

Self-healing Scenarios. One possible instance of our domain model is already given by our ACC example in Fig. 1. The ECUs *BrakeControl-ECU*, *EngineControl-ECU*, and *ACC-ECU* have a word size of 32-bit, while *LightControl-ECU* and *Infotainment-ECU* have a word size of 16-bit. The only software component needing just 16-bit running on a 32-bit ECU is *BrakeLight-SW* (cf. Fig. 3). Furthermore, we have a *LIN-Bus* which delivers input and feedback to the user via the infotainment system. There is also a *CAN-Bus* with more throughput for a fast delivery of signals from the time critical systems ACC,

engine control, and brake control. These two buses are connected by a gateway, which translates between the protocols and compensates the throughput differences. Since some software components also need a certain bus load, the possible throughput of these buses also influences the reconfiguration scenario.

One possible scenario is that the *ACC-ECU* crashes. In this case, we have to move the deployed software components *ObjectDetection-SW*, *VelocityControl-SW*, *Tracking-SW*, and *TimeGapControl-SW* to other ECUs. One condition for this reconfiguration is that every 32-bit component has to be deployed on a 32-bit ECU, and that 16-bit components can be deployed on 16-bit as well as 32-bit ECUs. The second condition is that these 32-bit ECUs have to be connected to the high-throughput *CAN-Bus*. The third and last condition in this case is that *Tracking-SW* and *TimeGapControl-SW* have to be deployed onto the same ECU, since the first one produces a lot of traffic for the second one such that even the high-throughput bus may not suffice.

After detecting the ACC-ECU's crash, the reconfiguration is started. The reconfiguration calculates a new possible and correct deployment for the software components on the ECUs. To realize this deployment different strategies are possible. One scenario is to use a planning algorithm to re-allocate the software components, similar to [13]. While the constraint solver is working (and the reconfiguration is done) the ACC functionality is deactivated and the driver is informed. After finishing the reconfiguration the system is started and the functionality is working again.

The reconfiguration has changed the system instance to the one depicted in Fig. 5. The software components *Tracking-SW* and *TimeGapControl-SW* are, as stated in the third condition, deployed together on *BrakeControl-ECU*. Since there were not enough resources on this ECU, the formerly deployed *BrakeLight-SW* requiring only a 16-bit ECU was moved to *Light-Control-ECU*. The remaining two components *VelocityControl-SW* and *ObjectDetection-SW* were moved to *Engine*

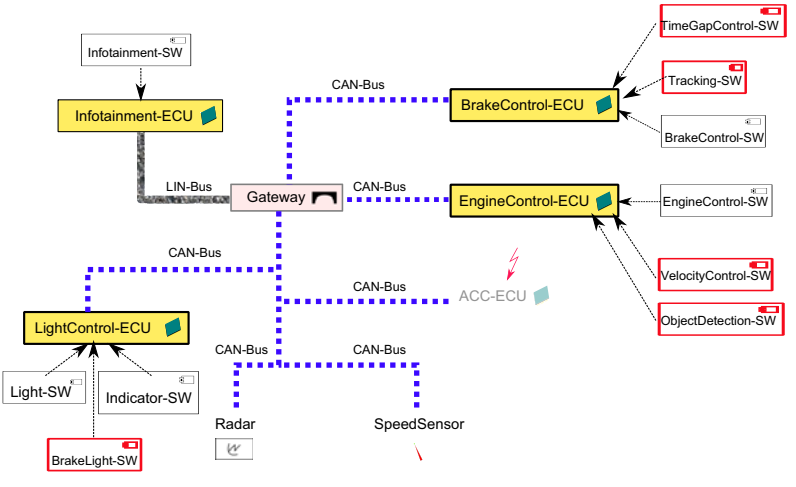

Fig. 5. The ACC system after the reconfiguration

Control-ECU. No component was moved to *Infotainment-ECU* because it is connected via the low-throughput *LIN-Bus*.

Further self-healing scenarios our approach could be applied to are the failures of sensors or software components. Newer versions of the ACC system use a combination of different sensor systems to retrieve more detailed data (e.g., camera and radar). If one part of such a sensor combination fails, the other part can resume its work with lower quality characteristics. This leads to a graceful degradation of the sensor system. In case of the failure of a software component the problem is more challenging, because backup components have to be provided on other ECUs. The current resource constraints in the automotive sector make it difficult to use redundancy, but there is a trend towards more powerful ECUs. This would support the self-healing ability of automotive systems to deal with failed software components, also.

5 Related Work

Other approaches also deal with the reconfiguration in automotive systems. The Dyscas project [4,3] specified a system architecture for automotive systems which also provides reconfiguration possibilities. These possibilities are mainly captured in the developed middleware. But the described reconfiguration is mainly used to recognize errors and to degrade the functionality, so that a minimal functionality is still there. In contrast our approach uses the reconfiguration to re-establish the entire functionality. By using the different constraints it is possible to calculate a new configuration for the whole system.

Another approach that deals with reconfiguration is described in [6]. There, a possibility is demonstrated how the AUTOSAR tool Systemdesk[3] can be extended to model reconfiguration possibilities. The system can then choose between the different reconfigurations at runtime. The problem with this approach is that every possible reconfiguration has to be modeled at design time and then be stored at the ECUs. The number of needed scenarios is exponential with the components and sensors in the system. To avoid that amount of data and the development work, we use the constraint solver to calculate a correct reconfiguration. Additionally we can also react to situations that nobody has thought about during the development phase.

In [20] an extension to the AUTOSAR standard is demonstrated that makes dynamical reconfiguration and self-healing possible. But this approach only takes some properties like memory into account. Other constraints like that two software components have to be deployed to the same node are missing. In our approach it is possible to specify such constraints. As already mentioned, we are currently working on a planner to rearrange the software components on the ECUs to attain the calculated correct system states. In [15] a PDDL planner for a robotic scenario is presented. It also uses constraints for defining the reconfiguration task. The tasks there are sequences of capabilities and it does not consider quantitative constraints like presented in this paper.

[3] www.dspace.de

[18] proposed a framework for self-healing and self-adaptive networks of hardware/software nodes was presented. It focuses on the restarting of jobs on other nodes in case of a failure. Properties like load balancing are addressed. In contrast to this work, they consider no dependencies between the particular jobs.

6 Conclusions and Future Work

We have shown that the ODP is applicable to data-flow systems and especially to automotive systems. With only minor adjustments and extensions the pattern was successfully applied to an adaptive cruise control system. Furthermore, we were able to show how to self-heal the system after an ECU failure simply by exploiting the constraints which are specified for the ODP-DFS.

The AUTOSAR standard supports the basic requirements for self-healing by propagating a component-based software design and appropriate interface definitions for decomposition. However, it currently does not support reconfiguration at runtime since all connections between components and hardware nodes are fixed prior to runtime. Nevertheless, there are ongoing efforts to extend AUTOSAR to support reconfiguration [20].

The application of the restore invariant approach to the constraints defined in this paper computes a new system configuration based on the response to failures. We did not address how to reconfigure the system from the old to the new configuration. Depending on the current system configuration, the desired system configuration and additional constraints, the reconfiguration maybe encompasses multiple and possibly concurrent steps. We are currently investigating whether planning as in [13] can be integrated into our approach for this purpose.

Finally, the presented ODP-DFS will be extensively evaluated to check whether all relevant characteristics of data-flow systems are contained in the pattern. This includes redundancy to support safety-critical systems and time to support real-time systems. In this area it is possible that extensions are necessary. The application to other domains than automotive systems is especially interesting.

References

1. Autosar specification (2009), `www.autosar.org`
2. Amor-Segan, M., McMurran, R., Dhadyalla, G., Jones, R.: Towards the Self Healing Vehicle. In: Automotive Electronics, 2007 3rd Institution of Engineering and Technology Conference on, pp. 1–7 (2007)
3. Anthony, R., Leonhardi, A., Ekelin, C., Chen, D., Trngren, M., de Boer, G., Jahnich, I., Burton, S., Redell, O., Weber, A., Vollmer, V.: A future dynamically reconfigurable automotive software system. In: Proceedings of the IESS (2007)
4. Anthony, R., Rettberg, A., Chen, D., Jahnich, I., de Boer, G., Enkelin, C.: Towards a dynamically reconfigurable automotive control system architecture. International Federation for Information Processing (IFIP) 231, 71–84 (2007)
5. Badros, G.J., Borning, A., Stuckey, P.J.: The cassowary linear arithmetic constraint solving algorithm. ACM Trans. Comput.-Hum. Interact. 8(4), 267–306 (2001)

6. Becker, B., Giese, H., Neumann, S., Schenck, M., Treffer, A.: Model-based extension of autosar for architectural online reconfiguration. In: Proceedings of the ACES-MB 2009, CEUR Workshop Proceedings, CEUR-WS.org, pp. 123–137 (2009)
7. Beckert, B., Keller, U., Schmitt, P.H.: Translating the object constraint language into first-order predicate logic. In: Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), pp. 113–123 (2002)
8. Brière, D., Favre, C., Traverse, P.: A family of fault-tolerant systems: electrical flight controls, from airbus a320/330/340 to future military transport aircraft. Microprocessors and Microsystems 19(2), 75 (1995)
9. Broy, M.: Challenges in automotive software engineering. In: International Conference on Software Engineering, ICSE (2006)
10. Grimm, K.: Software technology in an automotive company: major challenges. In: ICSE 2003: Proceedings of the 25th International Conference on Software Engineering, pp. 498–503. IEEE Computer Society, Washington (2003)
11. Güdemann, M., Nafz, F., Ortmeier, F., Seebach, H., Reif, W.: A specification and construction paradigm for organic computing systems. In: Brueckner, S.A., Robertson, P., Bellur, U. (eds.) SASO, pp. 233–242. IEEE Computer Society, Los Alamitos (2008)
12. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2006)
13. Klöpper, B., Meyer, J., Tichy, M., Honiden, S.: Planning with utilities and state trajectories constraints for self-healing in automotive systems. In: Proc. of the Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Budapest, Hungary. LNCS, Springer, Heidelberg (2010)
14. Nafz, F., Ortmeier, F., Seebach, H., Steghöfer, J.P., Reif, W.: A universal self-organization mechanism for role-based organic computing systems. In: González Nieto, J., Reif, W., Wang, G., Indulska, J. (eds.) ATC 2009. LNCS, vol. 5586, pp. 17–31. Springer, Heidelberg (2009)
15. Satzger, B., Pietzowski, A., Trumler, W., Ungerer, T.: Using automated planning for trusted self-organising organic computing systems. In: Rong, C., Jaatun, M.G., Sandnes, F.E., Yang, L.T., Ma, J. (eds.) ATC 2008. LNCS, vol. 5060, pp. 60–72. Springer, Heidelberg (2008)
16. Seebach, H., Nafz, F., Steghöfer, J.P., Reif, W.: software engineering guideline for self-organizing resource-flow systems. In: Proceedings of the Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (2010)
17. Seebach, H., Ortmeier, F., Reif, W.: Design and Construction of Organic Computing Systems. In: Proceedings of the IEEE Congress on Evolutionary Computation 2007. IEEE Computer Society Press, Los Alamitos (2007)
18. Streichert, T., Haubelt, C., Koch, D., Teich, J.: Concepts for Self-Adaptive and Self-Healing Networked Embedded Systems. In: Organic Computing. Springer, Heidelberg (2008)
19. Torlak, E., Jackson, D.: Kodkod: A relational model finder. pp. 632–647 (2007), http://dx.doi.org/10.1007/978-3-540-71209-1_49
20. Trumler, W., Helbig, M., Pietzowski, A., Satzger, B., Ungerer, T.: Self-configuration and self-healing in autosar. In: APAC-14 (2007)
21. Weiss, G., Zeller, M., Eilers, D., Knorr, R.: Towards Self-organization in Automotive Embedded Systems. In: González Nieto, J., Reif, W., Wang, G., Indulska, J. (eds.) ATC 2009. LNCS, vol. 5586, pp. 32–46. Springer, Heidelberg (2009)
22. Winner, H., Hakuli, S., Wolf, G.: Handbuch Fahrerassistenzsysteme- Kapitel Adaptive Cruise Control. Vieweg Verlag (2009)