2014 IEEE International Symposium on Software Reliability Engineering Workshops

Run-time Adaptive Error and State Management for Open Automotive Systems

Jelena Frtunikj*
*fortiss GmbH
Guerickestrasse 25
80805 München, Germany

Michael Armbruster†
†Siemens AG
Otto Hahn Ring 6
81739 München, Germany

Alois Knoll‡
‡Technische Universität München
Boltzmannstrasse 3
85748 Garching bei München, Germany

***Abstract*—Over the past few years semi-autonomous driving functionality was introduced in the automotive market and this trend continues towards fully autonomous cars. While in autonomous vehicles, data from various types of sensors realize the new highly safety critical autonomous functionality, the already complex system architecture faces the challenge of designing highly reliable and safe autonomous driving system. A common approach to build a reliable real-time system is using hardware replication; however the solution tends to be very costly. An alternative approach is providing support for adaptive error and state management and effective resource utilization that allows a system to adapt and reconfigure after failures of part of the system without requiring the user intervention. In addition, the end-customer is used to the possibility of easy personalization or extensibility of the electronic systems with new HW or SW. In this paper we present our model-based framework and run-time system that enables system extension and improves the safety of autonomous driving systems by providing reusable formal scheme enabling adaptive error and resource management. A case study explaining the applicability of the approach is presented.**

***Keywords*—dependability, safety, autonomous driving systems**

I. Introduction

Nowadays, the automotive industry faces the challenge to manage the electrics and electronics E/E-architecture's complexity [3] while in parallel more functionality (we call each considered software-implemented function "system function" herein) is added within a vehicle. A recent study [2] shows that the E/E architecture faces the challenge of raising demand for vehicle automation up to fully autonomous functionality (driving, parking). Future vehicles will be also able to cooperate (Car2Car) in order to perform many functions in a more effective and efficient way. Nowadays, the customer is used to the possibility of easy personalization and extensibility of electronic systems. The infotainment-domain has already demonstrated these capabilities and it drives a similar expectation now within the automotive domain. Thereby, the end-customer does not care about qualities such as dependability, since those are expected to be ensured by default.

Due to high criticality and the requirement for fail-operational behavior of the autonomous functions, the E/E architecture must provide built-in mechanisms to achieve fault-tolerance. Traditional fault tolerance techniques such as installing multiple identical hardware backup systems may be cost prohibitive, since automotive systems typically have tight cost constraints. Adaptive error and state management and efficient resource usage offers the possibility to increase system dependability without having to provide redundant system resources. The concept enables in case of a subsystems failure resulting in loss of some system resources, a run-time evaluation of the system state and a reconfiguration to be applied. The reconfiguration, provides the required functions with the highest safety and reliability demand.

A possible technical approach for adaptive error and sate management and efficient resource handling is a formal framework for specifying degradation rules and a run-time system (RTS) which ensures different non-functional qualities of interfaces and functionalities at run-time. The idea behind using a RTS approach that ensures the previously mentioned functionality is to reuse the already developed safety measures for different systems and functions and save future development costs spent on non-functional qualities. In addition, the RTS as a core element of a so called "open automotive architecture" also provides mechanisms for system extension.

The challenge considered here is providing a framework for generic error and state management applicable to all system functions. To do so, we consider each function as a composition of fault containment regions (FCR). Due to the fact that only the function developer has the knowledge which FCRs compose certain function and which system resources (e.g. CPU, memory) are required by the function, the information has to be given a priori as configuration parameters. Since the RTS, contains safety mechanisms that can determine the "health" state of each FCR and it has information about the available non-faulty system resources, it is able to identify the "health" and degradation of each function and to perform the needed reconfigurations. It is important to mention, we assume that functions can be performed in several degradation levels of service (depending on the quality of the subsystems that compose the function) and the proposed solution at run-time selects the highest possible, but still safe.

This article is structured as follows. In Section 2, we first introduce the target open fault-tolerant E/E architecture and give a short overview of its main features. Afterward, we give a detailed description of the proposed concept and we present a case study where the approach is applied. Section 4 compares gives brief overview of related work in the area. At the end we give a brief conclusion and summarize the future steps.

II. Target System Architecture

As stated in the introduction in order to enable automatic run-time integration of different sub-systems (HW or SW) or functions, new open E/E architecture that provides Plug&Play ability is required. To enable Plug&Play in the automotive field, the system architecture needs to provide capabilities to guarantee extra functional properties for resource utilization,

978-1-4799-7377-4/14 $31.00 © 2014 IEEE
DOI 10.1109/ISSREW.2014.77

safety and security. Such architecture is suggested in the Robust and Reliant Automotive Computing Environment for Future eCars[1] project and its basic principles have been already presented in [10]. The proposed platform is composed by a scalable set of central execution nodes (also called Duplex Control Computers (DCCs)) and a set of peripheral execution nodes providing the physical sensing and actuating. The DCCs assemble the Central Platform Computer (CPC) and are connected to each other and to the Smart-Aggregates by redundant switched Ethernet-Links. An RTS interconnects all system components and facilitates generic safety mechanisms such as real-time deterministic scheduling, data exchange services, health monitoring and diagnosis, as well as time and space partitioning for applications. The proposed system has two different power supplies, named red and blue. Each execution node is supplied by either the red or the blue power supply so in case one power-supply fails, only a subset of the nodes get lost and the residual nodes continue the operation.

To support Plug&Play, the traditional message-oriented approach is replaced by a data-centric one: instead of specifying sender-receiver relationships, the subsystems developers have to specify the component interfaces by a standardized data model. Based on this data model, the RTS establishes data paths between subsystems. The data-model can be seen as a data-base full of data-elements describing the vehicle's, the environment's or the driver's state. Integration of new subsystems in the above mentioned E/E architecture is done in 3 steps (Figure 1). Since the integration occurs during run-time, it has to be ensured that the system keeps operating according to its specification.

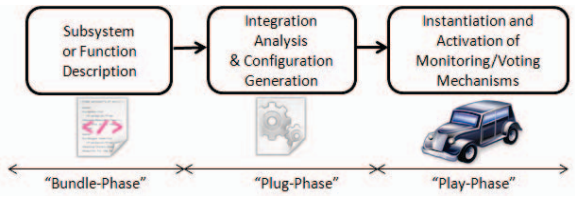

Fig. 1. Bundle- Phase, Plug- Phase, Play-Phase

The integration of new subsystem starts with a "Bundle-Phase" where the HW/SW subsystem is delivered with description information containing also the safety related information. The Plug&Play procedure is logically separated into two phases. In the "Plug-Phase" interpretation of description documents is done to re-plan the E/E configuration and adapted configuration data is provided. The next phase is entered only in case all required qualities especially those addressing functional safety can be ensured. In the "Play-Phase" the new configuration is enforced. For the remainder of this paper, we assume such architecture as the underlying basis.

III. Proposed Approach

This section presents the proposed concept by giving insights in our modeling approach and RTS.

A. Domain-specific meta-model

In order to be able to provide a generic approach dealing with error management, system function degradation and dynamic resource reconfiguration, we need to introduce changes to the development process. This means we have to take into account the abstraction of functions from their former dedicated electronic control unit and the requirement that functions can be integrated into a variety of architecture variants. Therefore, data dependencies between functions and the required CPU or memory resources need to be defined explicitly and unambiguously. To do so, at design time a domain-specific model has to be used. In Figure 2, a meta-model describing system functions, subsystems and their properties and dependencies, is shown. This is the first step towards automated and uniform function description. The model enables the composition of functions from different subsystems (HW and/or SW) and the definition of degradation rules. The degradation rules represent specification of reduced system function functionality after occurrence of failures of the subsystems from which the function is composed. The meta-model is used in a model-driven development tool which allows modeling each function based on the available data in the system and subsequently generating data structures from this model that are used at run-time by the RTS. The resulting models are called models@run-time [7] since they contain information (e.g. required memory resources, degradation rules etc.) that is relevant at run-time.

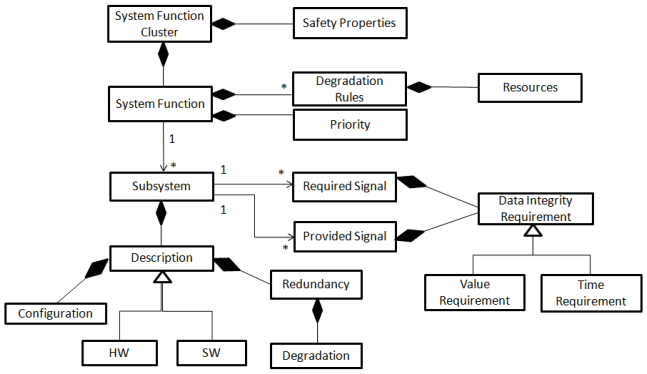

Fig. 2. Meta-Models defining function and its components

It is important to emphasize that each of the subsystems has configuration description that expresses additional restrictions (e.g. if the subsystem is a sensor, the description contains information as sensor position, viewing direction, maximum distance, type of target or collision objects etc.). The information is used to check if the data from one type of sensor in case of failure can be replaced by the data from another type of sensor. Moreover, the data integrity information is very important for the health monitoring mechanisms that provide the information required to determine the "health state" and degradation of the subsystem. By using the models@run-time, we provide the possibility for later system extension and safe integration of the subsystems. Whenever a new subsystem (HW or SW) is integrated in the system (Plug&Play) a description of it and its requirements have to be provided to the system. The idea behind the description is to establish a predefined and standardized component description of the single (sub-) systems that are to be integrated.

B. Formal System Model

Below we present the formal foundation of our approach as described in our work [4].

[1] Robust and Reliant Automotive Computing Environment for Future eCars, http://www.projekt-race.de/

Definition 1 A vehicle V = (F_a, SW_a, HW_a, D) is build up from a finite set of System Function Architecture F_a, a Software Architecture SW_a, a Hardware Architecture HW_a and a Deployment Configuration D.

Definition 2 A System Function Architecture $F_a = (S_f, S_{fc})$ is composed by a finite set of System Functions S_f and a set of System Function Clusters S_{fc}.

Definition 3 A System Function set $S_f = \{sf_1, ...sf_n\}$ contains the system functions of the vehicle. A system function can be realized by one or more SW components and the required Sensors and Actuators.

Definition 4 System functions are grouped in set of System Function Clusters $S_{fc} = \{sfc_1, ...sfc_k\}$, where $sfc_i \subseteq S_{fc}$ while $\forall i, j : sfc_i \cap sfc_j = \emptyset$. We define the mapping of $sf \in S_f$ to $sfc \in S_{fc}$ with $\epsilon(sfc) \rightarrow \{sf_i \in S_f | sf_i$ is mapped to $sfc\}$.

The grouping of sf is based on the safety properties of the functions such as: 1) criticality level of the function (in the automotive standard called Automotive Safety Integrity Level (ASIL)) and 2) performance requirements regarding fail-operational or fail-safe behavior. This way of grouping reduces the system complexity with regard to the amount of combinations to be considered for reconfiguration.

Definition 5 A Software Architecture is composed by a finite set of SW components $SW_a = \{s_1, ...s_m\}$ that belong to at least one system function $s \in SW_a$ to $sf \in S_f$ with $\alpha(s) \rightarrow \{s_i \in SW_a | s_i$ is mapped to $sf\}$.

Definition 6 A Hardware Architecture is composed by a finite set of HW components $HW_a = \{h_1, ...h_l\}$. The set is divided in set of execution nodes and set of peripheral actuator or sensor nodes $HW_a = HW_e \cup HW_p$.

Definition 7 The Deployment Configuration D = ($\delta(sfc)$) defines how the System Function Clusters and the corresponding SW components are deployed to the execution nodes HW_e. For $sfc \in S_{fc}$, we define to $\delta(sfc) \rightarrow \{h_i \in HW_e | sfc$ is executed to $h_i\}$.

The execution nodes represent the previously mentioned duplex control-computers (DCC).

Definition 8 A fault is a physical defect, an imperfection or a flaw that occurs within some hardware or software component. An error is the manifestation of a fault and a failure occurs, when the component's behavior deviates from its specified behavior [6].

Depending on the level of abstraction where a system is investigated, the occurrence of a malicious event may be classified as a fault, error or a failure. We define all malicious events that might occur within a subsystem as error.

Fault Tolerance deals with mechanisms (error and fault handling) in place. These mechanisms allow a system to deliver the required service in the presence of faults despite degraded level of that service.

Definition 9 A subsystem set is defined by the set of SW components and peripheral actuator or sensor nodes $SubS = SW_a \cup HW_p$.

Following definitions 3, 5, 6 and 9, a system function is unambiguously defined by a set of logical subsystems $SF = \{subS_1, ...subS_k\}$. The subsystems represent Fault-Containment Regions (FCR) which is seen as black box w.r.t. safety and error handling. The FCRs have precisely specified interfaces in the domains of time and value, which are required to detect anomalies at run-time. This means, in a case of error the FCR and with that the subsystem is marked and handled as faulty. The alteration of subsystem state can be expressed formally by the definition of new state transition $subSState_{ok} \rightarrow subSState_{err}$. This definition and handling is required in order to be sure that the fault within the FCR will not be extended out of the defined subsystem borders.

Definition 10 Based on the subsystem $subS$ "health" state (error free or erroneous) and the redundancy information, different degradation level of the subsystem $subSDeg^0$, $subSDeg^1$, ..,$subSDeg^N$ can be defined.

The subsystem degradation level (also named only degradation) can take values form 0 to N. The zero degradation level is the lowest one and represent fully functionality, while the Nth level is the highest one and means no functionality available (the subsystem is in erroneous state s_{err}).

Definition 11 A system function sf degradation predicate $sfDeg^x$ is a boolean function over a set of degradation level states of the subsystems composing the function. The set of system function degradation predicates represents the specification of the function and system degradation w.r.t. safety. For each degradation predicate a set of attributes $A = \{memoryResources, runtimeResources\}$ are specified and are used by the reconfiguration mechanism that keeps system safety after a failure of execution component. The system degradation predicate can also get values form 0 to N.

Reconfiguration mechanism: In case of execution nodes scarce (due to a failure), a reconfiguration mechanism has to be activated in the system in order to make the decision about which system functions to be run in the system and which not. The decision criteria needs to take into consideration the available resources (execution and system resources e.g. different sensors) and the criticalites of the functions. As this is obviously a computationally complex multi-dimensional optimization problem that has to be solved at run-time, techniques that are not computationally extensive like greedy approximation algorithm should be used.

C. Run-time system

Even though our approach is based on a formal foundation, we explain it in an informal way for a better understanding. The overall approach can be compared to the already existing autonomic computing paradigm, where elements of a system are managed by control loops based on the so-called **MAPE-K** (Monitor, Analyze, Plan, Execute - Knowledge based) cycle which optimizes operation of the supervised elements.

We consider a system function $sf \in S_f$ as a composition of subsystems. Since the subsystems have precisely specified interfaces in the domains of time and value, that information is used for configuring the safety mechanisms of the RTS, which provide the information and the error indications required to determine the "health" state of the corresponding subsystems. More detailed explanation of the safety mechanisms that the RTS offers can be find in [5]. We define the following relevant FCRs/subsystems of a system function: 1) sensors or actuators; 2) application software components - SW functions implementing the system function control algorithm including its resource partition (time and space) in which it is running. The defined FCRs also include the communication links for sending and/or receiving data. To enable calculation and appropriate

determination of function degradation level at run-time, a RTS component named System Function Manager (SFM) identifies the state, correct or faulty, of all subsystems. Figure 3 shows the process of fault detection (health monitoring), consolidation of error indications, ”health” state determination and the mapping to RTS components.

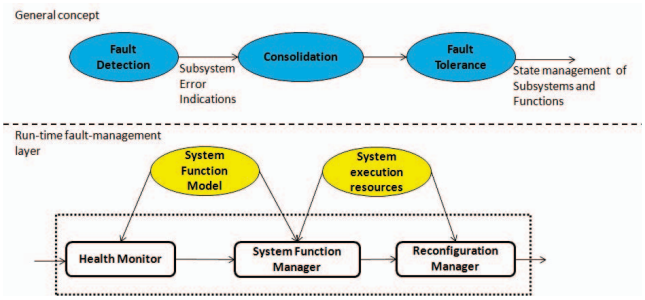

Fig. 3. Separation of fault detection from fault handling

Depending on the error indications that the health monitoring RTS components generate and the redundancy type of the subsystem (e.g. single, double, triple redundancy etc.), the SFM identifies the ”health” state and the degradation level of each subsystem at run-time. The task of the System Function Manager is mainly focused on the state management (possible states: isolated, passive and active) of all subsystems $subS$ belonging to all system functions, based on their current ”health” state and on the available system resources. The System Function Manager together with the Reconfiguration Manager determines the next state. In case of a permanent fault, the FCR and the corresponding $subS$ is isolated, in a case of transient fault the FCR is passivated and when no faults exists the FCR continues to be active.

Based on the ”health” states and the redundancy information, we have defined the following $subS$ degradation level:

- degradation level 0 ($subSDeg^0$): data available (no fault detected and the subsystem is ”active”)
- degradation level 1 ($subSDeg^1$): data available but data coming via one network link are not available (and the subsystem is ”active”)
- degradation level 2 ($subSDeg^2$): data available but one redundant subsystem from same type is faulty (meaning lost) (and the subsystem is ”active”)
- degradation level N ($subSDeg^N$): data are not available due to a fault (and the subsystem is ”isolated”)

The information about the degradation level of each subsystem is used to calculate the degradation of a system function $sfDeg$ at run-time. Since only the function developer has the knowledge and the expertise, which subsystems compose and are required for certain system function, he is responsible for defining the allowed degradation of the system function. A degradation rule for a system function is expressed by means of boolean algebra. The boolean expression includes all subsystems and their degradation state $subSDeg$. An example of such an expression for a system function consisting of three subsystems can look like:

$$sfDeg^1 = subSDeg_i^0 \wedge subSDeg_j^1 \wedge subSDeg_k^0$$

An example system function sf in a vehicle is pedestrian detection and auto brake function that consists of four subsystems/FCRs: camera $subS_{camera}$, radar $subS_{radar}$, brake $subS_{brake}$, and SW component implementing the algorithm for pedestrian detection $subS_{pdswc}$. Each of the $subS$ has different degradation levels depending on the redundancy constellation:

- camera: $subSDeg^0_{camera}$ and $subSDeg^N_{camera}$
- radar: $subSDeg^0_{radar}$, $subSDeg^1_{radar}$ and $subSDeg^N_{radar}$
- brake: $subSDeg^0_{brake}$ and $subSDeg^N_{brake}$
- SW component: $subSDeg^0_{pdswc}$ and $subSDeg^N_{pdswc}$

The degradation rules specified by the predicates define the dependency between the specific function and the sensors or actuators and other applications whose data is required in order the function to work. Based on the degradation rules and the actual degradation level $subSDeg_i^x$ of each subsystems, the boolean expressions are evaluated at run-time and the ”best” system function degradation $sfDeg$ is calculated. An example rule specified by the system function developer looks like:

$$sfDeg^1_{pedDet} = subSDeg^0_{camera} \wedge subSDeg^1_{radar} \wedge subSDeg^0_{brake} \wedge subSDeg^0_{pdswc}$$

A formal description of the degradation calculation algorithm, which calculates at run-time the actual system function degradation level for all active sf can be find in [4].

Various sensor types with different modalities are initially deployed or are added afterward in autonomous vehicle. Since sensors are prone to intermittent faults, using different sensor is better than duplicating the same type of sensors as different types of sensors typically react to the same environmental condition in diverse ways. With the above in mind, our approach for open automotive system offers the possibility to define additional degradation rules including different types of sensors, in case when different types of sensors, provide the same data/information. In such situation, a possibility to switch to a different source providing the same information (that has correct configuration w.r.t. position, viewing direction etc.) exists and the system function still remains fully available in the system. An example of such a case for the previously described function, is when data from Radar $subS_{radar}$ sensor can be ”substituted” by data produced by a LiDAR (Light detection and ranging) $subS_{lidar}$ sensor. Additional degradation rules for the pedestrian detection and auto brake functionality like the one below can be specified.

$$sfDeg^0_{pedDet} = subSDeg^0_{camera} \wedge subSDeg^0_{lidar} \wedge subSDeg^0_{brake} \wedge subSDeg^0_{pdswc}$$

Based on the fact that the function developer also specifies the required resources for each degradation level and the criticality level of the functions, the RTS has the possibility to react appropriately in case of resource scarce. For example, if not enough system resources are available, the RTS can deploy and run all high criticality functions in the full functionality, but the less critical ones in a degraded mode in which they require fewer resources. At run-time, based on the ”health” status of each subsystem, the ”lowest” available degradation level of each system function is calculated. Depending on the resources available in the system, the Reconfiguration Manager (RM) RTS component dynamically decides if and

at which degradation level to execute each function (details in next subsection). An Execution Manager component, which manages the scheduling and execution of functions, performs the required schedule changes.

D. Reconfiguration algorithm

In order to enable automatic graceful degradation at run-time, reconfiguring the software components belonging to sf of the system to accommodate the available hardware duplex control-computers upon detection of a fault has to be accomplished by the Reconfiguration Manager (by changing their state: activate or deactivate). The proposed algorithm dynamically selects software components of the system functions belonging to $sfc_i \subseteq S_{fc}$ that maximize the safety properties of the system. Our three step algorithm (Figure 4) is similar and can be compared to a Greedy approximation algorithm.

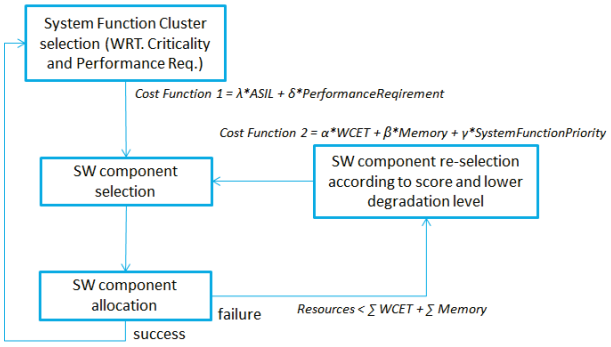

Fig. 4. Reconfiguration algorithm

In Step 1 we choose the system function cluster based on the *Cost Function 1* that should be present in the system in order to satisfy the safety properties (see meta-model in Figure 2). The cost function contains two variables (and two coefficients) that can influence the score of the function. The criticality variable $ASIL$ can get values $\{0, 1, 2, 3, 4\}$ for $\{QM, ASILA, ASILB, ASILC, ASILD\}$ criticality level respectively. The $PerformanceRequirement$ can be assigned values $\{1, 2\}$ for $\{fail - safe, fail - operational\}$ behavior respectively. First the clusters with the highest *cost*, meaning ASIL D and a performance requirement ($fail - operational$) is selected. For the two coefficients holds $\lambda + \delta = 1$, and it is up to the system architect to decide upon their weight. In our case study we have assigned 0,5 value to both coefficient. In case of $fail - operational$ (with hot-standby slave) requirement, the redundant sfc have to be allocated to two different computing units connected to different power supplies to avoid that both instances are lost simultaneously when a power supply fails.

Step 2 selects the SW components that belong to the system functions of the selected cluster. Step 3 then ensures that all selected SW components are allocated to one of the available execution units (DCC) and that constraints (enough memory and CPU resources) for a valid configuration hold. Failure of the allocation results in a re-execution of previous step, with SW components that have smaller *Cost Function 2* as a result of degradation. The degradation can be decided based on the allowed degradation levels of the system functions. Normally, the degraded levels require less hardware resources. As shown on Figure 4, the score is dependent on the three parameters and their corresponding weight coefficients. Despite of the required resources, the priority of the function, which states the importance (w.r.t. system safety) of the system function within the cluster (which can have values in range $[0 - M]$ depending on the system specification), also has influence on the score of the *Cost Function 2*. Again, for the three coefficients holds the following $\alpha + \beta + \gamma = 1$. In our case study we have assigned 0,2 value to α and β, and 0,6 to γ since the system architect has the opinion that the priority of the function within the cluster has a greater influence on system safety.

E. Case study

This subsection presents how the explained concept is used in a scenario, where the pedestrian detection and auto brake system function and other system functions are integrated in a demonstration vehicle. Firstly, the function developer specifies different degradation rules describing the pedestrian function. In addition to the rules, for each of them the requirements regarding the run-time execution resources (worst-case execution time- WCET) and the RAM and ROM are stated in ms and $MBytes$ respectively. The pedestrian detection and auto brake function belongs to one cluster $\epsilon(sfc_1) = \{sf_1\}$. Two other lusters sfc_2, sfc_3 also exsist in the system (Figure 5). The sfc_2 that contains the system functions $\epsilon(sfc_2) = \{sf_2, sf_3, sf_4\}$ has the highest *Cost Function 1 = 3.0*, since it is highly critical cluster with $ASILD$ and fail-operational requirement. The third cluster is specified with $\epsilon(sfc_3) = \{sf_5\}$.

System Function Cluster	System Functions	SW Component	CostFunction1 (ASIL, PerformanceRequirement)
SFC1	SF1 :Pedestrian Detection and Brake	S1: PedDetect	2,5 (ASILD/fail-safe)
SFC2	SF2: Driving, SF3: Steering, SF4: Braking	S2: ManDriving S3: ManSteering S4: ManBraking	3 (ASILD/faill-op)
SFC3	SF5: Infotainment	S5: Infotainment	0,5 (QM/faill-safe)

Fig. 5. Case Study: System Function Clusters

All five system functions $S_f = \{sf_1, sf_2, sf_3, sf_4, sf_5\}$ and their corresponding $SW_a = \{s_1, s_2, s_3, s_4, s_5\}$ are allocated to the four computing units DCCs as shown on the left side of Figure 6. Since the $sfc2$ cluster has a fail-operational requirement, the SW components s_2, s_3, s_4 belonging to the corresponding system functions of that cluster, are allocated twice (on two different DCCs). We can notice that all SW components belonging to $sfc2$ are allocated to the same computing unit as our algorithm suggests, and s_1 is also allocated to the same DCC. When the first fault in the system happens and the third DCC computing unit (and its resources) is not available any more, the software components s_2, s_3, s_4 belonging to the highly critical cluster sfc_2 are able to get the resources which are available on the fourth computing DCC (middle part of Figure 6) using our reconfiguration algorithm. When a second fault occurs in the system and the resources from the first DCC are not available any more, our framework decides upon reconfiguration and reallocation of the software components belonging to the sfc_2 to the second computing unit, where enough resources are available. Since the sfc_1 has a higher *Cost Function 1* then sfc_3, the software component s_5 will not be executed any more due to unavailability of resources, and s_1

will be reallocated to the second computing unit. However, due to unavailability of sufficient computing resources the system function sf_1 will degrade in the $sfDeg^2_{pedDet}$ level where less computing resources are required.

Fig. 6. Case Study: Reconfiguration

IV. Related Work

Here, we compare our approach against available solutions provided by industry and scientific community.

In [9], authors show an approach to analyze graceful degradation. They use a utility function to measure the set of active features which can be seen as quite similar to our cost functions. To reduce complexity, they group components by defining subsystems and system as a composition of subsystems that each contribute to overall system utility. We see systems as a composition of system function that are then composed from subsystems. We group the system functions into clusters according to their safety requirements. The main difference is that they focus only on the reconfiguration algorithm but not on RTS and the reconfiguration mechanisms.

Authors in [11], suggest an approach that includes formal specification technique to describe known standard fault tolerance solutions. They propose fault tolerance patterns (similar to our degradation rules) which capture the essential structure and relevant deployment restrictions of these solutions. Fault tolerance patterns are easily applied during the design of component-based systems to increase the reliability or availability of specific components. However, they do not aim at self-reconfiguration and extension of the system at run-time.

Current industry practice dealing with faults and failures in embedded systems focuses on the traditional approaches of fault tolerance and fault containment [8], as we do. SW subsystems are physically separated into different hardware modules. Additionally, system resources, such as sensors and actuators, that may be commonly used are replicated for each subsystem. Similar to our framework, this approach provides assurance that faults will not propagate between subsystems since they are physically partitioned, but on contrast the fault tolerance is achieved only by replicating resources and subsystems rather than shedding some not that critical functionality.

The automotive AUTOSAR standard [1] describes a platform which allows implementing vehicle applications and minimizes the barriers between functional domains. One of the main objectives of AUTOSAR version 4 is to support safety critical applications. To do so, the AUTOSAR execution environment offers safety capabilities that focus on the correct execution of software components only, and the monitoring of functional behavior of the system functions is neglected. Currently 3 levels of statically pre-configured mode managers allowing degradation are supported by AUTOSAR. However, they lead to a cluttered and complex implementation. In comparison to this we enable easy system and function degradation by specifying intuitive degradation rules, so our approach can be seen as an extension and improvement to AUTOSAR. In addition, we also support the possibility of system extension which currently AUTOSAR does not support.

V. Conclusion and Outlook

This paper presented a domain-specific meta-model that enables to compose different high-level functions from different components, define their dependencies and their required resources. We specified RTS components that based on this information are able to calculate the available degradation level of all system functions, and based on the resources and the information about the criticality of the functions, decide upon appropriate reconfiguration. Future work includes further implementation and integration of our approach in the ”Revolution” car demonstrator of the RACE project. In addition, we will evaluate how the proposed architecture can be used in cooperative vehicular systems with Car2Car technologies.

Acknowledgment

The work presented here is partially funded by the German Federal Ministry for Economic Affairs and Energy (BMWi) through the RACE project.

References

[1] AUTOSAR Group. AUTomotive Open System ARchitecture (AUTOSAR) Release 4.1, 2013.

[2] M. Bernhard et al. *The Software Car: Information and Communication Technology (ICT) as an Engine for the Electromobility of the Future*. ForTISS GmbH, March 2011.

[3] M. Broy, I. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, 2007.

[4] J. Frtunikj, V. Rupanov, M. Armbruster, and A. Knoll. Adaptive Error and Sensor Management for Autonomous Vehicles: Model-based Approach and Run-time System. In *4th International Symposium on Model Based Safety Assessment*, October 2014.

[5] J. Frtunikj, V. Rupanov, A. Camek, C. Buckl, and A. Knoll. A safety aware run-time environment for adaptive automotive control systems. In *Embedded Real-Time Software and Systems (ERTS2)*, February 2014.

[6] J. C. Laprie, A. Avizienis, and H. Kopetz, editors. *Dependability: Basic Concepts and Terminology*. 1992.

[7] G. Lehmann, M. Blumendorf, F. Trollmann, and S. Albayrak. Meta-modeling runtime models. In *MoDELS Workshops*, pages 209–223, 2010.

[8] J. Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, 1999.

[9] C. Shelton, P. Koopman, and W. Nace. A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems. In *Object-Oriented Real-Time Dependable Systems (WORDS)*, pages 156–163, Jan 2003.

[10] S. Sommer, A. Camek, K. Becker, C. Buckl, A. Zirkler, L. Fiege, M. Armbruster, G. Spiegelberg, and A. Knoll. Race: A centralized platform computer based architecture for automotive applications. In *Electric Vehicle Conference (IEVC), 2013 IEEE International*, pages 1–6, Oct 2013.

[11] M. Tichy and H. Giese. Extending fault tolerance patterns by visual degradation rules. In *Workshop on Visual Modeling for Software Intensive Systems (VMSIS)*, pages 67–74, 2005.