Adaptive Error and Sensor Management for Autonomous Vehicles: Model-based Approach and Run-time System

Jelena Frtunikj[1], Vladimir Rupanov[1], Michael Armbruster[2], and Alois Knoll[3]

[1] fortiss GmbH, An-Institut Technische Universität München,
Guerickestr. 25, 80805 München, Germany
{frtunikj,rupanov}@fortiss.org
[2] Siemens AG, Corporate Research and Technologies,
Otto-Hahn-Ring 6, 81730 München, Germany
{michael.armbruster}@siemens.com
[3] Fakultät für Informatik, Technische Universität München
Boltzmannstrasse 3, 85748 Garching bei München, Germany
{knoll}@in.tum.de

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. Since sensors are prone to intermittent faults, using different sensors is better and more cost effective than duplicating the same sensor type because of diversity of reaction of different sensor typesto the same environmental condition. Specifying and validating sensors and providing technical means that enable usage of data from different sensors in case of failures is a challenging, time-consuming and error-prone task for engineers. Therefore, in this paper we present our model-based approach and a run-time system that improves the safety of autonomous driving systems by providing reusable framework managing different sensor setups in a vehicle in a case of a error. Moreover, the solution that we provide enables adaptive graceful degradation and reconfiguration by effective use of the system resources. At the end we explain in an example when and how the approach can be applied.

Keywords: safety, sensor models, autonomous driving systems, adaptive graceful degradation

1 Introduction

Nowadays, the automotive industry faces the challenge to manage the electrics and electronics E/E-architecture's complexity [12] while in parallel more and more functionality (we will call each considered software-implemented function

"system function" herein) is added within a vehicle. Moreover, a recent study [9] shows that the E/E architecture faces the challenge of raising demand for vehicle automation up to fully automated driving. Therefore, the new E/E architecture must be scalable enough to support autonomous functionality, such as driving [10], parking or charging, and new driver assistance systems.

Due to high criticality and the requirement for fail-operational behavior of these functions, the E/E architecture must provide built-in mechanisms to achieve fault-tolerance. This means that systems should be able to resume affected functions without negligible interruption. Traditional fault tolerance techniques, such as installing multiple identical hardware backup systems, may be cost prohibitive for automotive systems. This introduces limits to the design effort and redundant resources that can be spent to make the system dependable. Graceful degradation mechanisms provide increased system dependability without need for providing redundant system resources. It enables, in case of a subsystem failure, resulting in the loss of some system resources, run-time evaluation of the system state and reconfiguration to be applied. The reconfiguration is required for efficient utilization of the remaining resources and different sensor/actuator modalities to execute the required functions.

A possible technical approach to enable graceful degradation consists of a formal framework for specifying degradation rules and a run-time system that ensures different non-functional qualities of interfaces and function behavior at run-time. The idea behind using a run-time system approach is to reuse the already developed safety measures for different systems and functions and save future development costs spent on non-functional aspects. This run-time system should implement technical means that can ensure fault-tolerance and enable adaptive graceful degradation of automotive system functions by effectively utilizing system resources and available data (supplied by different sources, e.g. sensors). However, the technical measures that enable the graceful degradation of functions must be generic, in order to be easily reused for all system functions in the vehicle.

The challenge considered in this paper is to provide a framework for generic (function-unspecific) graceful degradation applicable to all system functions that takes into consideration different types of data sources (sensors), their configuration and the available system resources. 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 etc.) are required by the function, the information has to be provided a priori as configuration parameters. Since the run-time system contains safety mechanisms that are capable of determining the "health" state of each FCR and has information about the available non-faulty system resources, it is able to identify/diagnose the "health" and degradation level of each function and to perform the necessary reconfigurations in order to provide the required functionalitiy in the system every moment. The reconfiguration is based on the available system resources, subsystems and their configuration, function criticality and function resource requirements. Our

approach uses a formal function model and a set of formal constraints that describe the validity of possible function degradation. The approach (and the models that we use) allows to analyze at run-time if the desired system safety properties can be fulfilled and which set of functions should still be provided after one or multiple isolation FCRs or system resources.

This article is structured as follows. In Section 2, we first introduce the target scalable fault-tolerant E/E architecture and give a short overview of its main features. Afterwards we give a detailed description of the proposed concept, explaining the meta-model and the run-time environment in details. Section 4 presents an example explaining how this approach can be applied. Section 5 compares our approach against available solutions provided by industry and scientific community. The last section provides a brief conclusion and summarizes future steps.

2 RACE system architecture and safety concept

As discussed in the introduction, future vehicles must support an increasing amount of new complex functionalities, such as predictive advanced driver assistance systems (ADAS) up to highly and fully automated driving and parking. Since, today's vehicle E/E architectures are very complex, extending the system with more hardware or software (which supports new functionalities) is not trivial at all. As a result, a new system architecture for modern cars is required.

Basic requirements for such an E/E architecture are to be scalable, open and thus easily expandable. Such architecture is proposed in the Robust and Reliant Automotive Computing Environment for Future eCars (RACE)[4] project [13]. The fail-operational centralized platform (Figure 1) consists of a redundant and reliable communication infrastructure based on a switched Ethernet topology, redundant power supply (blue and red lines in Figure 1) and redundant controllers. The centralized platform computer is composed from two or more duplex control-computers (DCC) and is responsible for executing all system functions. In order to guarantee fail-safe behavior, a DCC has two execution lanes that monitor input and output data mutually. In case an error occurs their results are discarded. Fail-operational behavior is guaranteed by a second DCC, which takes over the control tasks in case the first DCC has failed. The sensors and actuators used in the approach are smart and responsible for the low level control tasks. They are connected to the DCCs by a high-bandwidth Ethernet network.

A run-time system, together with the operating system, drivers, and components provides services, interconnects all system components. The run-time system facilitates generic safety mechanisms such as real-time deterministic scheduling, data exchange services, health monitoring and diagnosis, as well as an execution platform with time and space partitioning for applications running on the same HW and having different Automotive Safety Integrity Level (ASIL) classifications.

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

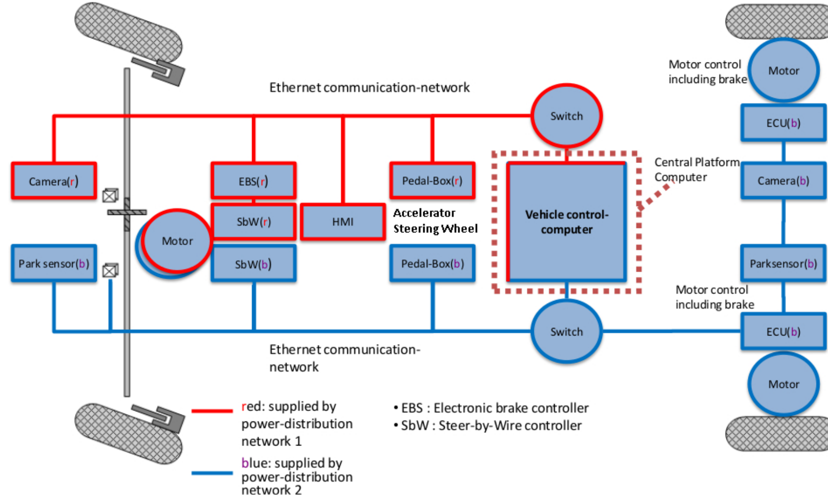

Fig. 1: System Architecture: fail-operational design

3 Proposed approach

This section presents the proposed concept by giving insights in our modeling approach and the run-time system.

3.1 Domain-specific meta-model

In order to provide a generic approach dealing with system function degradation and dynamic resource reconfiguration, we need to introduce changes to the development process. 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 (we consider any combination of allocated functions to the control/computing units of the system as a variant). Therefore, data dependencies between functions, required sensor data and resource requirements (e.g., CPU and memory) need to be defined explicitly and in an unambiguous manner. To do so, at design time a domain-specific model is used. In Figure 2, a meta-model describing system functions, subsystems and their properties and dependencies, is depicted. This is the first step towards automated and uniform function description. The model enables composition of functions from different subsystems (HW and/or SW) and definition of degradation rules. The degradation rules represent specification of reduced functionality of a system function 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 available data in the system and subsequent generation of data structures from this model for further run-time use by the run-time system. The resulting models are called models@run-time [2] since they contain information (e.g. required memory resources, degradation rules etc.) that is relevant at run-time.

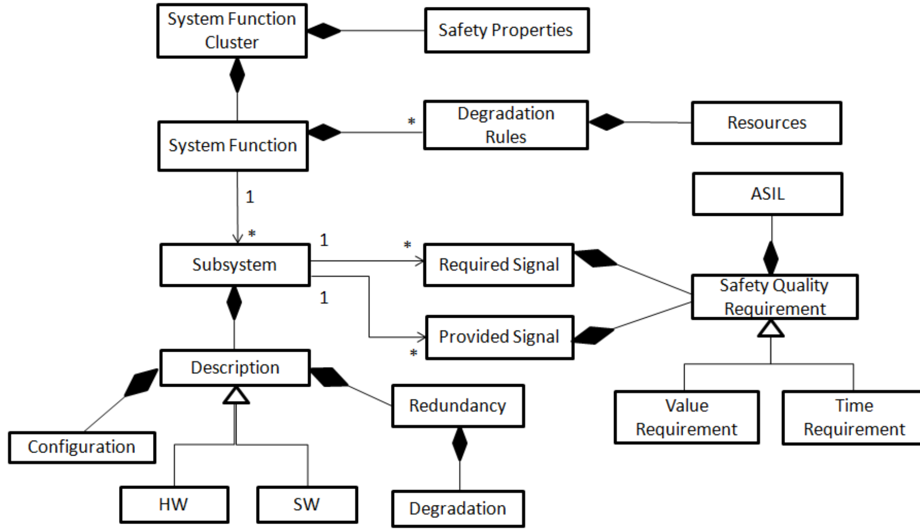

Fig. 2: Domain-specific meta-model defining function and its components

It is important to emphasize that each subsystem has a configuration description that expresses additional restrictions to be considered in the models. For example, if a subsystem is a sensor, the configuration description (Figure 3) contains such information as sensor position, viewing direction, maximum distance, type of target or collision objects w.r.t. geometry and material data, etc. [3]. The information is used to identify and validate allowed sensor configurations and check if data from one type of sensor in case of failure can be replaced by the data from another type of sensor.

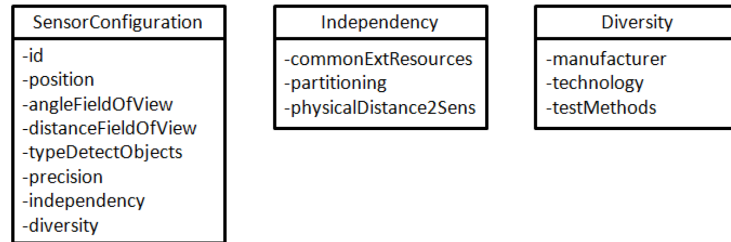

Fig. 3: Meta-Model defining sensor configuration

Moreover, each subsystem fulfills the assigned ASIL capability according to ISO 26262 [11]. In order to check if data from different sensors (considered and modeled as subsystems) can be fused to achieve certain higher ASIL level, properties such as diversity and independence have to be specified (Figure 3). Two subsystems are diverse in case they are produced by different manufacturers, use different technology and production processes, use different test methods etc. Independence property can be checked for example by checking partitions

of functions or software elements, physical distance between hardware elements, common external resources, etc.

The information about the safety quality requirement is important for the health monitoring mechanisms that provide information required to determine the "health state" and degradation of the subsystem and thereby of a function.

3.2 Formal System Model

Below we present the formal foundation of the models used in our approach.

Definition 1 A vehicle $V = (F_a, SW_a, HW_a, D)$ is built 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 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 into a 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 $\epsilon\,(sfc) \rightarrow \{sf_i \in S_f | sf_i \text{ is mapped to } sfc\}$.

The grouping of sf is based on the safety properties of the functions such as: 1) criticality level of the function (ASIL); 2) performance requirements regarding fail-operational or fail-safe behavior. This way of grouping of the system functions reduces the system complexity with regard to the amount of combinations to be considered for deployment.

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 $\alpha\,(s) \rightarrow \{s_i \in SW_a | s_i \text{ is mapped to } sf\}$.

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 $\delta\,(sfc) \rightarrow \{h_i \in HW_e | sfc \text{ is executed to } h_i\}$.

The execution nodes represent the previously mentioned duplex control-computers (DCC). The set of the execution nodes is also called Central Platform Computer (CPC).

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 [1].

Depending on the level of abstraction, at which a system is explored, 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 $SubS = SW_a \cup HW_p$.

Lemma 1 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 can be seen as black boxes 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.

Each subsystem $subS$ has a defined configuration $subSConfig$. The configuration information is taken into consideration when an evaluation about interchangeable subsystems is performed. In case of a failure of one $subS$, the data required from that $subS$ can be substituted by the before validated interchangeable subsystem.

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. A system function 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.

3.3 Run-time system and degradation approach

Even though our approach is based on a formal foundation, here we explain the approach in more details in an informal way for the sake of clarity.

As mentioned before, 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 run-time system, which provide information and error indications required to determine the "health" state of the corresponding subsystems. We define the following relevant FCRs/subsystems of a system function: 1) sensors or actuators and 2) application software components - SW functions implementing the system function control algorithm, including respective partitions (in both time and space domains). The defined FCRs also include the communication links, through which they send and/or receive data.

In order to enable calculation and appropriate determination of function degradation level at run-time, a run-time system component named System Function Manager (SFM) identifies the state of each subsystem belonging to each system function. This means the SFM is able to determine the "health" state (correct or faulty) and the degradation level of all subsystems based on the system state and the error indications that result from diagnosis of these specific subsystems. More detailed explanation of the safety mechanisms that the run-time system offers can be found in [5]. The process of fault detection (health monitoring), consolidation of error indications, "health" state determination and the mapping to run-time system components is depicted on Figure 4.

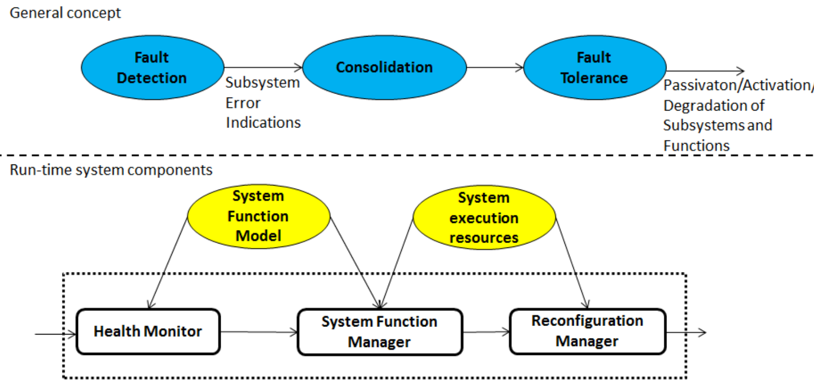

Fig. 4: Separation of fault detection from fault handling

Depending on the error indications that the health monitoring run-time system 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. SFM is based on a developed algorithm that evaluates the error indications, which are a result of detected faults, and in that way it provides the required information that represents the

basis for error handling. The task of the System Function Manager is mainly focused on the phases of isolation, passivization and activation on of all subsystems $subS$ belonging to all system functions, based on their current "health" state. The System Function Manager together with the Reconfiguration Manager determine the next "health" 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 if no faults more are found, the FCR continues to be active. The mentioned "health" states can be applied to all types of subsystems (FCRs) that are defined in the system.

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

- 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 in the previously mentioned system architecture are not available (and the subsystem is "active")
- degradation level 2 ($subSDeg^2$): data available but one redundant subsystem from same type is faulty (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/she 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 (sensors, actuators and/or SWC) looks like following:

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

An example system function sf in a vehicle is pedestrian detection and auto brake function, which 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 these subsystems 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 $subSDeg^N_{brake}$
- pedestrian detection SW component: $subSDeg^0_{pdswc}$ and $subSDeg^N_{pdswc}$

The degradation rules specified by the predicates define the dependency between a 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 subsystem, the boolean expressions are evaluated at run-time and the "best" system function degradation $sfDeg$ is calculated. For the pedestrian detection and auto brake function the system function developer has specified the following degradation rules:

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

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

$$sfDeg^2_{pedDet} = subSDeg^0_{camera} \wedge subSDeg^N_{radar} \wedge subSDeg^0_{brake} \wedge subSDeg^0_{pdswc}$$

......

$$sfDeg^N_{pedDet} = \neg(sfDeg^0_{pedDet} \vee sfDeg^1_{pedDet} \vee sfDeg^2_{pedDet} \vee sfDeg^3_{pedDet} ... \vee sfDeg^{N-1}_{pedDet})$$

Based on the the degradation calculation algorithm shown in Algorithm 1, the actual system function degradation level is calculated at run-time for all active sf belonging to all active system function clusters sfc.

Algorithm 1 Degradation calculation

```
1: for all SFCluster sfc in SFClusterList sfcList do
2:   for all SysFunction sf in SFCluster sfc do
3:     for all SubSystems sbubS in SysFunction sf do
4:       calculateHealthStateAndDegLevel(subS,subS.Deg)
5:     end for
6:     calculateSysFDegLevel(sf,sf.degRules,sf.subSs)
7:   end for
8: end for
```

As stated in [8], autonomous vehicles have various sensor types with different modalities. Since sensors are prone to intermittent faults, using a different sensor is better than duplicating the same type of sensors. Different types of sensors typically react to the same environmental condition in diverse ways. For example, in case a vehicle is equipped with radars for blind spot detection, if rear-looking radar does not work properly, a software algorithm detecting obstacles from images obtained via rear-looking camera can be used.

With the above in mind, our approach offers the possibility to define additional degradation rules including different types of sensors in cases when different types of sensors provide the same data/information. This is beneficial especially in the case of a failure of one or more data sources. In such a 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. The correct configuration of two interchangeable sensor subsystems is validated in our model-driven development tool and the information is available at run-time. For example, properties like angle of view ais checked:

$$subSConfig_{lidar}.angleFOV >= subSConfig_{radar}.angleFOV$$

$$subSConfig_{lidar}.distanceFOV >= subSConfig_{radar}.distanceFOV$$

An example of such a case for the previously described pedestrian detection and auto brake 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. We have to mention that the different sensor might require different monitoring functions that help at run-time the "health" state of the sunsystem to be calculated. Additional degradation rules for the pedestrian detection and auto brake functionality like the one below can be specified. Having this opportunity is useful and important since various algorithms using different types of sensors even for a common goal may consume significantly different amount of resources.

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

The approach also offers the possibility to model data redundancy not only from same types of units but also from diverse ones. Different degradation levels of the units, are supported in case of diverse units and additional degradation rules for the function can be specified. The advantage of using data from both types of units lies in the possibility for data fusion and obtaining more thereby accurate information. With that also a higher ASIL level can be achieved. In case of the above example, both radar $subSDeg_{radar}$ and LiDAR $subSDeg_{lidar}$ degradation can be included in the degradation rules of the function:

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

Based on the fact that the function developer also specifies the required resources for each degradation level and the criticality level of the functions (as shown in the meta-model Figure 2), the run-time system has the possibility to react appropriately in case of resource scarce. For example, if not enough system resources are available, the run-time system 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. The mentioned decision is done by the Reconfiguration Manager (RM) component that dynamically decides if and at which degradation level to execute each function. An Execution Manager component, which manages the scheduling and execution of functions, performs the required schedule changes.

3.4 Approach benefits

To sum up, we enable adaptive graceful degradation for all system function in a generic and uniform manner. Our degradation specifications have several advantages. The specifications: 1) are high-level, ensuring that the user is not overwhelmed by implementation details. Our specifications require users only to describe desired behavior, not implement techniques for achieving it; 2) are concerned only with describing functional behavior, yet they provide a natural

interface to the models used to describe the failures and degradation. Since the system function developer only has to focus on the description of each function instead of developing technical measures that manage graceful degradation, we save his efforts and time, by providing a run-time system that does that automatically for him. Furthermore, the formalized specification and the usage of generated models@run-time guarantee consistency and completeness in the critical transition from the requirements engineering to software design, where lot of errors can be introduced into the system by using conventional, non-formal techniques.

4 Example

This section presents how the previously explained concept can be used in a scenario, where pedestrian detection and auto brake system function in a vehicle is composed from data coming from different sensors.

The pedestrian detection and auto brake technology usually relies on data that come from camera, radar and/or LiDAR subsystems. Depending on the available data sources the function requires different amount of resources to detect pedestrians in the nearby environment of the vehicle. With the help of the model-based development tool, the developer of the function specifies the degradation rules for the different subsystem combinations and the different resource requirements for each degradation level of the function. In our case, the degradation rules stated in the previous section reflect the different possibilities for describing the function. The function has a $sfDeg^0_{pedDet}$ when all subsystems are fully available ($subSDeg^0_i$). The function degrades to second level $sfDeg^2_{pedDet}$ when the radar subsystem is not available any more $subSDeg^N_{radar}$. However, since in the system the data coming from the LiDAR sensor (whose configuration has been validated before) can substitute the data from the radar sensor the pedestrian detection and auto brake function can be run again at $sfDeg^0_{pedDet}$. This is important for autonomous vehicles where this functionality is very essential and contributes to the overall system safety.

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. For example, for $sfDeg^0_{pedDet}$ where we have the full functionality the resource requirements are the following ones: $WCET = 800ms$, $RAM = 10MBytes$ and $ROM = 10MBytes$. In comparison to that for $sfDeg^2_{pedDet}$ the $WCET = 500ms$, $RAM = 5MBytes$ and $ROM = 5MBytes$. This information is useful in case of computing resources scarce, since a function can be executed in a degraded mode where it (normally) requires less resources.

5 Related work

In this section, we discuss the related work and we present ones that are most relevant to our work and state the differences between the approaches.

Tichy [7], suggest an approach that includes formal visual 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. In contrast to our approach, they do not aim at self-reconfiguration of the system at run-time and do not offer any solution for sensor management.

A MDE approach for managing different sensor setups in a cyber-physical system development environment to leverage automated model verification, support system testing, and enable code generation is presented in [4]. The models are used as the single point of truth to configure and generate sensor setups for system validations in a 3D simulation environment. This approach only focuses on the validation process and the verification of possible pin assignments for connecting the required sensors and does not offers a run-time system that enables usage of the data from different sensors and a possibility for degradation.

Authors in [14] aim at graceful degradation by adapting the functionality of a system to the driving situation and the available resources. Since adaptation significantly complicates the development of embedded systems, they present an approach to the model-based design of adaptive embedded systems that allows coping with the increased complexity posed by adaptation. Furthermore, they show how the obtained models can be formally verified by a model checker. This approach is similar to ours, but in our opinion it is applicable at design time.

The industrial automotive AUTOSAR standard [15] describes a platform which allows implementing future vehicle applications and minimizes the current barriers between functional domains. One of the main objectives of AUTOSAR version 4 release is to support safety related applications by implementing features to comply with the safety ISO 26262 standard requirements. The AUTOSAR execution environment safety capabilities 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 that allow degradation are supported by AUTOSAR. However they lead to a cluttered and complex implementation. In comparison to this with our framework we enable easy and reusable system and function degradation by specifying intuitive degradation rules, so our approach can be seen as an extension and improvement to AUTOSAR.

6 Conclusion and future work

Motivated by new challenges that automotive architectures face, we presented a domain-specific meta-model that enables to compose different high-level functions from different components, define their dependencies and their required resources. Furthermore, we specified run-time system components that, based on this information. are able to calculate the available degradation level of all system functions at run-time, and based on the resources and the information about the criticality of the functions, perform appropriate reconfiguration (self-repair in case of failures).

Acknowledgment

The work presented in this paper is partially funded by the German Federal Ministry for Economic Affairs and Energy (BMWi) through the project Robust and Reliant Automotive Computing Environment for Future eCars.

References

1. Laprie, J.C. C. and Avizienis, A. and Kopetz, H.: Dependability: Basic Concepts and Terminology. Springer-Verlag New York, Inc. (1992)
2. Lehmann, G. and Blumendorf, M. and Trollmann, F. and Albayrak, S.: Meta-modeling Runtime Models. MoDELS Workshops (2010)
3. Roth, E. and Dirndorfer, T. and Kilian v. Neumann-Cosel and Fischer, M.-O. and Ganslmeier, T. and Kern, A. and Knoll, A.: Analysis and Validation of Perception Sensor Models in an Integrated Vehicle and Environment Simulation. Proceedings of the 22nd Enhanced Safety of Vehicles Conference (2011)
4. Mamun, M. ; Berger, C. ; Hansson, J.: MDE-based Sensor Management and Verification for a Self-Driving Miniature Vehicle. Proceedings of the 13th Workshop on Domain-Specific Modeling (2013)
5. Frtunikj, J. and Rupanov, V. and Camek, A. and Buckl, C. and Knoll, A.: A Safety Aware Run-Time Environment for Adaptive Automotive Control Systems. Embedded Real-Time Software and Systems (ERTS2) (2014)
6. Shelton, C.P. and Koopman, P. and Nace, W.: A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems. Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems (2003)
7. Tichy, M. and Giese, H.: Extending Fault Tolerance Patterns by Visual Degradation Rules. Proceedings of the Workshop on Visual Modeling for Software Intensive Systems (VMSIS) (2005)
8. Urmson, C. et al.: Autonomous driving in urban environments: Boss and the Urban Challenge. Journal of Field Robotics Special Issue on the 2007 DARPA Urban Challenge, Part I (2008)
9. Bernhard, M. et al.: The Software Car: Information and Communication Technology (ICT) as an Engine for the Electromobility of the Future, Summary of results of the "eCar ICT System Architecture for Electromobility" research project sponsored by the Federal Ministry of Economics and Technology (2011)
10. Dmitri, D. et al.: Path Planning for Autonomous Vehicles in Unknown Semi-structured Environments. Sage Publications, Inc. (2010)
11. International Organization for Standardization: ISO/DIS 26262 - Road vehicles. Functional safety. Technical Committee 22 (ISO/TC 22) (2011)
12. Broy, M. and Kruger, I.H. and Pretschner, A. and Salzmann, C.: Engineering Automotive Software, Proceedings of the IEEE (2007)
13. Sommer, S. et al.: RACE: A Centralized Platform Computer Based Architecture for Automotive Applications. Vehicular Electronics Conference (VEC) and the International Electric Vehicle Conference (IEVC) (2013)
14. Adler, R. and Schaefer, I. and Schuele. T.: Model-Based Development of an Adaptive Vehicle Stability Control System, Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen (MBEFF) (2008)
15. AUTOSAR Group: AUTomotive Open System ARchitecture (AUTOSAR) Release 4.1 (2013)