RACE RTE: A Runtime Environment for Robust Fault-Tolerant Vehicle Functions

Klaus Becker, Jelena Frtunikj, Meik Felser, Ludger Fiege, Christian Buckl, Stefan Rothbauer, Licong Zhang, Cornel Klein

RACE RTE: A Runtime Environment for Robust Fault-Tolerant Vehicle Functions

Klaus Becker*, Jelena Frtunikj*, Meik Felser†, Ludger Fiege†, Christian Buckl*,
Stefan Rothbauer†, Licong Zhang‡ and Cornel Klein†
*fortiss GmbH, Guerickestr. 25, 80805 Munich, Germany
†Siemens AG, Corporate Research and Technologies, Otto-Hahn-Ring 6, 81730 Munich, Germany
‡Institute for Real-Time Computer Systems, TU Munich, Arcisstr. 21, 80333 Munich, Germany
Email: * name.surname@fortiss.org; † name.surname@siemens.com; ‡ name.surname@tum.de

***Abstract*—The degree of automated operation in vehicles is increasing continuously. Manufacturers want existing and new functions to be integrated, which drives engineering costs. On the other hand, customers grow accustomed to a steady flow of new functionality on smart phones, partially integrated into their vehicles. In this paper, the Runtime Environment (RTE) of the RACE project is presented. Based on a cross-domain system topology, the RTE executes real-time applications of mixed criticality up to fail-operational behavior. It offers communication and safety mechanisms that are configurable in-field to support Plug&Play scenarios. Since integrated functions often require access to different vehicle domains, the vehicle runtime and configuration data model is reified in the RTE to enable test and verification of all these mechanisms.**

I. Introduction

The number of complex functions of high-end vehicles is steadily increasing. As a result, the complexity of the IT architecture implementing these functions has reached a level that is increasingly difficult to manage. Upcoming functions such as connected mobility and autonomous driving will further increase the functional complexity and safety requirements. Hence, it is necessary to rethink today's E/E architectures.

Within the RACE project (Robust and Reliant Automotive Computing Environment for Future eCars) such an E/E architecture was developed. The RACE platform provides generic configurable services covering functional safety requirements up to ASIL-D. The generic safety mechanisms simplify the development of complex vehicle functions and their integration on the platform. With scalable computing capacity that can be increased retroactively, the RACE system is equipped to meet the increasing demand for processing power to enable a long and attractive service life for vehicles. The integration of new application software or hardware may also happen after sale.

In this paper, we describe the core concepts of the RACE Runtime-Environment (RTE). The RTE provides inherent platform mechanisms for qualities such as safety. It therefore interconnects software applications, hardware sensors and vehicle actuators. App-developers for instance do not have to take care on handling probabilistic faults, since these are dealt within the platform, which takes appropriate counter measures. Instead, the developers have only to select the right level of safety both with respect to the required ASIL and availability (e.g. fail-operational). App-developers are supported in creating modular independent application components by the concept of data-centrism and a topic-based publish/subscribe description of interfaces. In this paper, we show the RTE mechanisms applied to a steer-by-wire example application, which we integrated into a RACE demonstrator car that we constructed [1]. In the long term, however, also more complex functions like highly automated driving may be deployed on top of the platform.

II. Foundations of the RACE Platform

A. RACE Platform Architecture

The RACE platform is a new system architecture for the information and communication infrastructure of future vehicles. It introduces a logically centralized platform computer that executes functions like control loops on vehicle level, fusion of different sensor values, situation detection and strategy planning of driver assistance functions. The *Central Platform Computer* (CPC) is connected to smart sensors and smart actuators (together called smart *aggregates*) with an Ethernet-based network. Smart aggregates run closed-loop controls of local physical processes (e.g. inverter of e-engines) and provide pre-processed raw sensor data. The CPC works on vehicle level and is responsible for vehicle-wide decisions and control algorithms that need or affect more than one aggregate.

The CPC consists of a scalable set of multiple so called *Duplex-Control-Computers* (DCCs) with each of them performing the calculations on two parallel computation paths (called *lanes*) to ensure high integrity. The software core for each of the DCC lanes is the middleware called *RACE RTE*. It handles the redundant data streams and checks their integrity.

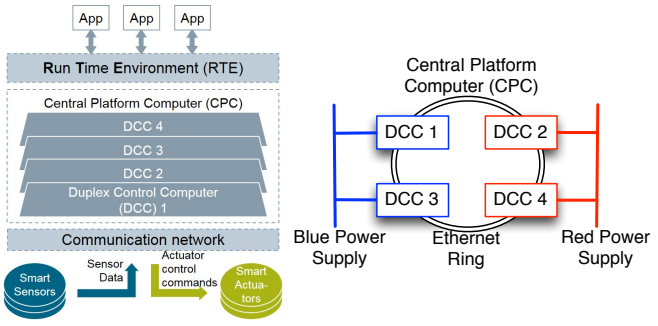

Fig. 1. The RACE platform

Fig. 1 shows an example CPC with four DCCs, connected with an Ethernet ring and having two different power supplies.

B. Scheduling, Networking and Partitioning

The RACE system works in fixed time-triggered real-time execution cycles. In each cycle a DCC receives data from other components via the RACE Ethernet, executes the scheduled applications and sends data to other DCCs or aggregates. The RACE network employs a scalable ring-based full-duplex switched Ethernet architecture, where an inner ring connects the DCCs of the CPC (cf. Fig 1) and outer rings or branches connect aggregates to the CPC. Frame preemption is used to reduce network transport delay for time-critical frames by allowing them to preempt the transmission of non-time-critical frames. This mechanism is currently being standardized in IEEE802.1Qbu (Time-Sensitive Networking, TSN) [1]. Further details of the network communication are provided in [2].

Spatial and temporal separation of mixed critical applications is ensured by running the RTE on top of the PikeOS[2] operating system and using partitioning mechanisms of PikeOS.

C. Integrity and Reliability

Depending on the use case and the system safety requirements, the system needs to provide different levels of integrity and reliability. In order to ensure data *integrity*, the RTE is able to execute applications on the two lanes of a DCC and check their results for bitwise equivalence. This bitwise comparison of lane results checks a DCC as fault-containment region to avoid any failure propagation in the platform. *Reliability* is guaranteed with fail-operational behavior in master-slave configurations, where a second DCC takes over in case of failures (hot or cold standby). Further DCCs can be employed to scale capacity or increase the mission time.

Aggregate redundancy (virtual layer)	Sensor redundancy (logical layer)	Communication link redundancy (physical layer)
single dual	single dual triple	single-link dual-link

TABLE I. VARIETY OF AGGREGATE REDUNDANCY

Redundancy of the CPC is complemented by a variety of aggregate redundancies on different layers (Table I). We distinguish the physical, logical and virtual layer. The aggregate redundancy (virtual layer) defines how many instances of a physical aggregate exist. The sensor redundancy (logical layer) defines how many sensors are present for one physical information per aggregate. The communication link redundancy (physical layer) defines with how many Ethernet connections an aggregate is connected to the rest of the system.

III. RTE ARCHITECTURE

The RACE RTE is a modular layered architecture and contains inherent features for error detection and error handling. The RTE runs on all DCCs and a tailored version is used on the aggregates. Some basic principles about the RTE have been already presented in [3] and [4]. To some extent, the RACE RTE is based on the Chromosome Middleware (XME) [5] and reuses some of its components.

[1]http://www.ieee802.org/1/pages/tsn.html
[2]http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept

In this paper, we go more into detail about certain RTE concepts such as data flow between RTE components, runtime error management and configuration of the RTE. Dedicated RTE layers are present for certain purposes, such as Network Management layer (for Frame packing/unpacking, CRC handling, etc.), Data Monitoring and Fusion layer, Indication Management as well as Application and Platform State Management. One cross cutting feature of the RTE is the *Test System*, which allows monitoring and manipulation of all data, status information and error indications during runtime without disturbing the system behavior, allowing diagnosis as well as fault-injection. The basic architecture and data-flow between the RTE components is shown in Fig. 2.

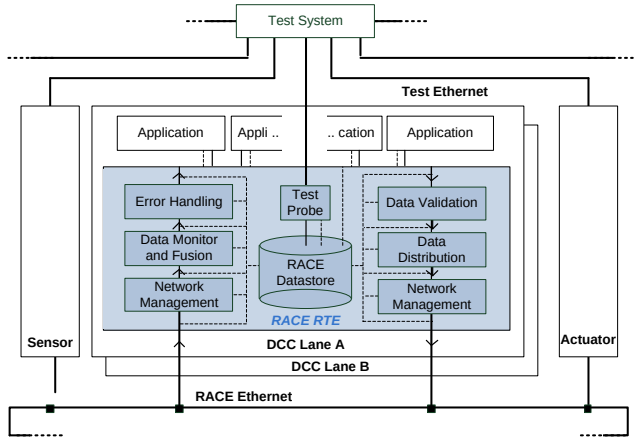

Fig. 2. Data-Flow through the RTE

A. Example Vehicle Function

As an example function, in this paper we consider a steer-by-wire application. A steering application runs on a DCC of the CPC, subscribes a steering-wheel-angle topic and publishes a control topic for the steering rack aggregate. Due to posed safety requirements, both steering aggregates are setup redundantly. The steering wheel data is sensed by a dual-triple/dual-link aggregate, meaning that two physical aggregates exist, each having three sensors. Hence, the angle of the steering wheel is sensed overall six times. Both aggregates publish their data. In the CPC, a steering application is executed (also redundantly as Master and Slave instance on two DCCs), which subscribes the sensed steering wheel angle. The RTE checks the sensor values for validity, masks erroneous values and finally fuses the values to one single value, which is given to the steering app. Finally, the steering application computes a control value for the also redundantly existing steering rack aggregates. This application architecture is shown in Fig. 3.

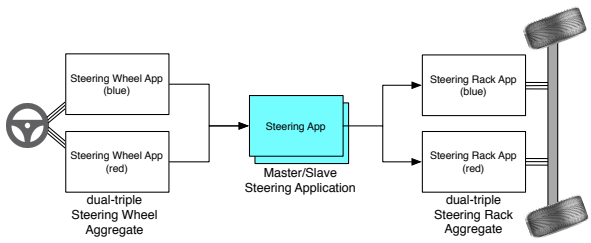

Fig. 3. Example Steering Application Architecture

B. Data Flow and Fault-Tolerance

The RACE RTE provides configurable fault-tolerance mechanisms that enable safe integration of safety-critical automotive functions, such as the steering function. Moreover, by decoupling the functionality from the safety mechanisms, we achieve a clear separation between the control steering functionality and the non-functional features. The mechanisms are grouped in three steps: error detection, error recovery, and fault treatment. Error detection raises *Error Indications* on the various data processing levels: Ethernet frames (physical aggregates, e.g. CRCs, timeouts), sensor values (logical aggregates, e.g. range checks), and fusioned data (virtual aggregates, e.g. deviations, inconsistencies). Based on configurable thresholds for each aggregate's indications, last-valid data or default values can be used for error recovery. As fault treatment, the RTE isolates erroneous fault-containment regions and supports fail-over to redundant hot-standby slaves or to more basic variants of applications (normal-law / direct-law fail-over).

We explain the input data-flow and the fault tolerance mechanisms within the RACE RTE for the steering application example (Fig. 4). The Network Management layer checks the CRC of the messages delivered by the steering aggregates and whether it was received in the correct time slot. Then one of the two dual-link messages is selected and unpacked. After the triple sensor data of the redundant steering wheel aggregates is unpacked, the monitors on logical level check for plausibility and consistency of each data instance. Here, checks w.r.t. validation of data against a predefined range of expected values are performed, and in case of an error the corresponding error indication is set (Fig. 4, red and green circles on logical level). The triple data from each aggregate is fused on the Data Monitoring and Fusion layer. On virtual layer, aggregate and data redundancies are hidden from the applications by a voting mechanism that selects and delivers the error-free data. In the case of the two triple aggregates, the data from the error-free aggregate is selected by the voting mechanism (Fig. 4, left aggregate). Only valid data is forwarded to the applications. Moreover, a quality information in form of a *green/yellow/red* indication is attached to the data, which is given to the applications. The steering-wheel sensor data gets a yellow quality indicator, since one of the redundant aggregates was detected as faulty. In case of a red indication, the last good known or a default value is forwarded.

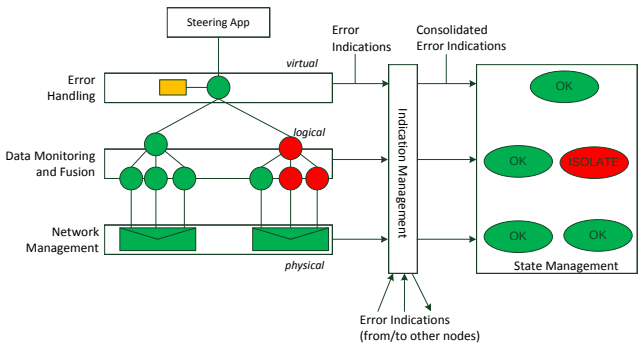

Fig. 4. Data-Flow and Fault-Tolerance within the RTE

The *Indication Manager* component collects all error indications. Its main function is the aggregation and inference of these indications to produce *Consolidated Error Indications*. The error indication set is also cyclically exchanged and synchronized with other nodes in the RACE network in order to achieve a consistent view on the system as a whole. The consolidated error indications are then made available to the *State Management* components (*Application Management* and *Platform State Management*). These RTE components provide a unified state management. The benefit of this kind of state management is the always deterministic state of all system components (aggregates, application software components and execution nodes) and with that the known state of the complete system, which is essential w.r.t. safety. A system component state is changed to *isolated* when an error is detected in order to prevent further error propagation. The state management components also trigger recovery actions (restart, activate a redundant copy, or shutdown) in case of failures in the system.

When two or more DCCs redundantly execute a control program such as the steering application then, for sake of unambiguous control, at each time point only one of these application instances is allowed to send signals to the connected actuators. This unambiguous control is handled by the state management components, which take care that only the data of the steering application instance that is marked as a *master* is forwarded to the steering rack. If the master becomes faulty, the RTE handles the fail-over switching and therewith the slave becomes the new master and the old master becomes isolated.

C. Non-Intrusive Fault-Injection Tests

The RACE architecture enables non-intrusive fault injection [6]. This is possible due to several measures such as exclusively reserved time slots, memory areas and network bandwidth for test actions. In addition, each DCC lane contains a system datastore that captures system-relevant data flows within and among RTE components. The RTE contains a *test probe* component, which enables monitoring of data and injection of faults at exact location and cycles. The test probe operates in a cyclical manner like all RTE components. In this way, the test probe is able to deterministically monitor system data (including error indications and state data) accumulated in the last cycle and to manipulate system data for the next cycle. For example, the test probe is able to inject a fault in one of the redundant aggregates by overwriting a sensor value with an invalid value. In accordance to the injected fault, a test is executed that checks the effectiveness of the RTE safety mechanisms that tolerate the corresponding fault. Such fault injection tests support both the development and certification of the systems. A complementary part to the test probes is the VITE[3] system, which efficiently runs the fault injection tests that spread over several nodes of the RACE system by instructing the test probes on each node.

D. Application Development and Configuration of the RTE

RACE application software components as well as physical aggregates are delivered with self-describing information contained in *manifests*. A manifest contains all data that is required to integrate a software component or an aggregate into a vehicle design. We use a RACE specific extension of the CHROMOSOME Modeling Tool (XMT) [5] to describe these manifests. For instance, the user has to define component

[3] Verification and Integration Testing Environment, www.aviotech.de

interfaces in form of publications and subscriptions of topics and attached attributes, as well as the *Worst Case Execution Time* (WCET) of the cyclic executable function and safety relevant information such as the component's requirement to behave fail-operational. The set of possible topics and attributes in one system is predefined in a so called *dictionary*. Topics may be for instance physical properties (temperature, pressure, etc.) or system data (recognized objects in front of the vehicle, a trajectory, etc.). Attributes describe the instances of a topic, such as the location and meaning of a temperature and the unit of measurement. The dictionary concept allows data compatibility between different applications, without requiring additional synchronization of interfaces.

The XMT tool offers to analyze virtual compositions of manifests in integrated product models. During the analysis, e.g. the logical data-flow between software components and aggregates is checked for completeness and unambiguity [7]. The analysis can be done in the tool for early integration prototyping, but is also done again by the RTE itself during RTE configuration. We also provide analysis of valid initial deployments of software-components to DCCs and possibly required graceful degradation in case of DCC failure scenarios [8]. Finally, manifest files are generated by the XMT design tool together with wrappers of the software components. In the wrappers, all read/write operations for the modeled interfaces are already present, as well as an initialization function and a function which is executed by the RTE in each cycle. Into this template, the developer can easily integrate its application behavior, for instance code generated from Matlab/Simulink.

The RTE configuration may happen at system start-up based on the set of given manifests, or later during a Plug&Play scenario. The logical data-flow is checked again as well as the fulfillment of all safety relevant properties. In case of conflicts or other problems, the configuration process is aborted and the integration is rejected. After a successful analysis, finally the configuration data structures for the RTE components are created and the RTE is put into operation.

IV. Related Work

RACE extends the idea of AUTOSAR [9] with a blueprint system architecture and generic safety mechanisms to enable fail-operational features. In RACE, configurable services support changes of deployed functions and thus Plug&Play extensions of vehicles with new software-based functions and new physical aggregates. Reifying the configuration of the platform, instead of tailoring the RTE for the deployed applications, introduces a certain runtime overhead, but is the key for changing the system in field. With the help of the runtime datastore, RACE supports non-intrusive fault injection tests that are impossible with AUTOSAR, since AUTOSAR lacks a build-in module comparable to the *test probe*.

RACE uses the data-centric design known from standards such as DDS [10] to achieve decoupling of functions and hence Plug&Play capability. Compared to DDS, RACE adds safety mechanisms and guarantees w.r.t. extra-functional properties.

Other related work, like Integrated Modular Avionics (IMA), ARINC 653, SIMATIC, Avionics Full DupleX Switched Ethernet (AFDX), FlexRay, TTEthernet and Audio Video Bridging (AVB) have been considered in [3].

V. Conclusion and Future Work

In this paper, we have shown the core concepts of the RACE Runtime Environment of a new robust E/E architecture for electric vehicles. The RACE RTE offers built-in fault-tolerance mechanisms like error detection and isolation of faulty components and non-intrusive fault-injection tests. A data-centric design provides independent development of modular application software components and physical aggregates. Manifests provide self-descriptions of software components and physical aggregates, which are used to analyze their composability and determine a valid configuration of the RTE. The application of redundancy mechanisms and runtime fail-over switching supports fail-operational features. We illustrated the RTE mechanisms on a steer-by-wire application.

As future work, we work in various directions for instance on improving the configuration process for Plug&Play, providing open models for representing environment information and methods and tools for agile development of vehicle functions. Furthermore, we want to introduce additional fault-detection mechanisms, such as plausibility functions, reducing resource footprint and supporting heterogeneous replication (diversity) of hardware and software in order to cover systematic faults. Modular and flexible data fusion mechanisms are tackled in project SADA (http://www.projekt-sada.de/en-sada).

Acknowledgment

This work is partially funded by the German Federal Ministry for Economic Affairs and Energy (BMWi) under grant no. 01ME12009, project RACE (http://www.projekt-race.de).

References

[1] M. Buechel *et al.*, "An automated electric vehicle prototype showing new trends in automotive architectures," in *IEEE 18th International Conference on Intelligent Transportation Systems (ITSC)*, 2015.

[2] M. Armbruster, L. Fiege, G. Freitag, T. Schmid, G. Spiegelberg, and A. Zirkler, "Ethernet-Based and Function-Independent Vehicle Control-Platform: Motivation, Idea and Technical Concept Fulfilling Quantitative Safety-Requirements from ISO 26262," *Adv. Microsystems for Automotive Applications (AMAA)*, pp. 91–107, 2012.

[3] S. Sommer, A. Camek, K. Becker, C. Buckl, A. Knoll, A. Zirkler, L. Fiege, M. Armbruster, and G. Spiegelberg, "RACE: A Centralized Platform Computer Based Architecture for Automotive Applications," in *IEEE Vehicular Electronics Conference / Int. Electric Vehicle Conference (VEC-IEVC)*, 2013.

[4] 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)*, 2014.

[5] C. Buckl, M. Geisinger, D. Gulati, F. Ruiz-Bertol, and A. Knoll, "Chromosome - a run-time environment for plug & play-capable embedded real-time systems," in *Workshop on Adaptive and Reconfigurable Embedded Systems (APRES)*, 2014.

[6] J. Froehlich and R. Schmid, "Architecture for a Hard-Real-Time System Enabling Non-intrusive Tests," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2014, p. 24.

[7] K. Schorp and S. Sommer, "Component-Based Modeling and Integration of Automotive Application Architectures," in *IEEE International Electric Vehicle Conference (IEVC)*, 2014.

[8] K. Becker and S. Voss, "Analyzing graceful degradation for mixed critical fault-tolerant real-time systems," in *IEEE 18th International Symposium on Real-Time Distributed Computing (ISORC)*, 2015.

[9] "AUTomotive Open System ARchitecture (AUTOSAR) Release 4.2," AUTOSAR Group, October 2014.

[10] "Data-distribution-service," http://www.omg.org/spec/DDS, Object Management Group (OMG).