Fault-Tolerant Deployment of Real-Time Software in AUTOSAR ECU Networks

Kay Klobedanz[1], Jan Jatzkowski[1], Achim Rettberg[2], and Wolfgang Mueller[1]

[1] University of Paderborn/C-LAB, 33102 Paderborn, Germany
{kay.klobedanz, jan.jatzkowski, wolfgang.mueller}@c-lab.de
[2] Carl von Ossietzky University Oldenburg, 26129 Oldenburg, Germany
achim.rettberg@uni-oldenburg.de

Abstract. We present an approach for deployment of real-time software in ECU networks enabling AUTOSAR-based design of fault-tolerant automotive systems. Deployment of software in a safety-critical distributed system implies appropriate mapping and scheduling of tasks and messages to fulfill hard real-time constraints. Additional safety requirements like deterministic communication and redundancy must be fulfilled to guarantee fault tolerance and dependability. Our approach is built on AUTOSAR methodology and enables redundancy for compensation of ECU failures to increase fault tolerance. Based on AUTOSAR-compliant modeling of real-time software, our approach determines an initial deployment combined with reconfigurations for remaining nodes at design time. To enable redundancy options, we propose a reconfigurable ECU network topology. Furthermore, we present a concept to detect failed nodes and activate reconfigurations by means of AUTOSAR.

1 Introduction

Today's automotive vehicles provide numerous complex electronic features realized by means of distributed real-time systems with an increasing number of electronic control units (ECUs). Many of these systems implement safety-critical functions, which have to fulfill hard real-time constraints to guarantee dependable functionality. Furthermore, subsystems are often developed by different partners and suppliers and have to be integrated. To address these challenges, the AUTomotive Open System ARchitecture (AUTOSAR) development partnership was founded. It offers a standardization for the software architecture of ECUs and defines a methodology to support function-driven system design. Hereby, AUTOSAR helps to reduce development complexity and enables smooth integration of third party features and reuse of software and hardware components [1]. Figure 1 illustrates the AUTOSAR-based design flow steps and the resulting dependencies for the deployment of provided software components [2]. Deployment implies task mapping and bus mapping resulting in schedules affecting each other. The problem of mapping and scheduling tasks and messages in a distributed system is NP-hard [3]. Beside hard real-time constraints, safety-critical systems have to consider additional requirements to guarantee dependability. Hence, deterministic communication protocols and redundancy concepts

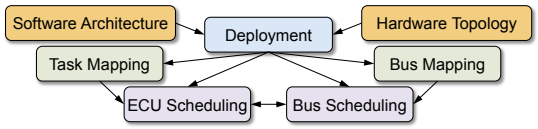


Fig. 1. System design flow steps and their dependencies [2]

shall be utilized to increase fault tolerance of such systems [4]. AUTOSAR supports FlexRay, which is the emerging communication standard for safety-critical automotive networks. It provides deterministic behavior, high bandwidth capacities, and redundant channels to increase fault tolerance. To further increase fault tolerance, node failures shall also be compensated by redundancy.

We present an approach for real-time software deployment built on AUTOSAR methodology to design fault-tolerant automotive systems. Our approach determines an initial deployment combined with necessary reconfigurations and task replications to compensate node failures. The determined deployment solution includes appropriate task and bus mappings resulting in corresponding schedules that fulfill hard real-time constraints (cf. Fig. 1). In addition, we propose a modified version of a reconfigurable ECU network topology presented in [5] to enable flexible task replication and offer the required redundancy. Regarding AUTOSAR, we propose a flexible Runnable-to-task mapping for fault-tolerant systems and present a concept for an AUTOSAR-compliant integration of our fault-tolerant approach: We propose an AUTOSAR Complex Device Driver (CDD) to detect failed nodes and initiate the appropriate reconfiguration.

The remainder of this paper is structured as follows. After related work and an introduction to AUTOSAR we present our proposal for a reconfigurable ECU network topology in Section 4. Section 5 describes our fault-tolerant deployment approach and applies it to a real-world application before we introduce a concept for AUTOSAR integration in Section 6. The article is closed by the conclusion.

2 Related Work

In general, scheduling of tasks and messages in distributed systems is addressed by several publications [6–8]. Other publications propose heuristics for the design of FlexRay systems [9–11]. In [12] strategies to improve fault tolerance of such systems are described. However, we propose an approach for fault-tolerant deployment of real-time software specific for AUTOSAR-based design flow. AUTOSAR divides task mapping into two steps: Mapping (i) software components (SWCs) encapsulating Runnables onto ECUs and (ii) Runnables to tasks that are scheduled by an OS. Since the number of tasks captured by AUTOSAR OS is limited, Runnable-to-task mapping is generally not trivial [13]. Although some approaches solve one [14] or even both [15] steps for an AUTOSAR-compliant mapping, to our knowledge, [16] is the only one considering this combined with fault tolerance. But unlike our approach, only a subset of the software requires hard real-time and each redundant Runnable is mapped to a separate task.

3 AUTOSAR

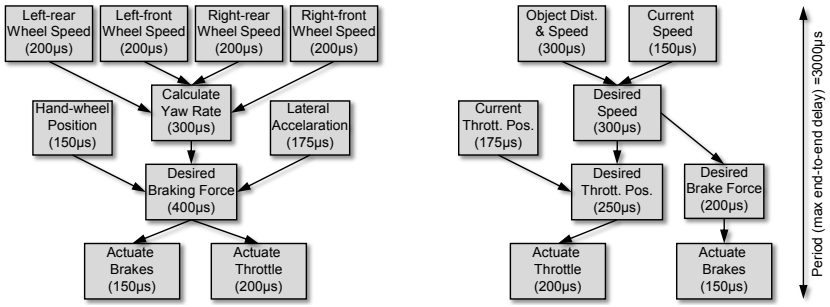
AUTOSAR provides a common software architecture and infrastructure for automotive systems. For this purpose, AUTOSAR distinguishes between *Application Layer* including hardware-independently modeled application software, *Runtime Environment (RTE)* implementing communication, and *Basic Software (BSW) Layer* providing hardware-dependent software, e.g. OS and bus drivers.

An Application Layer consists of *Software Components* (SWCs) encapsulating complete or partial functionality of application software [17]. Each Atomic-SWC has an internal behavior represented by a set of Runnables. "Atomic" means that this SWC must be entirely – i.e. all its Runnables – mapped to one ECU. Runnables model code and represent internal behavior. AUTOSAR provides RTE events, whose triggering is periodical or depends on communication activities. In response to these events, the RTE triggers Runnables, i.e. RTE events provide activation characteristics of Runnables. Based on RTE events, all Runnables assigned to an ECU are mapped to tasks scheduled by AUTOSAR OS. AUTOSAR *Timing Extensions* describe timing characteristics of a system related to the different views of AUTOSAR [18, 19]. A timing description defines an expected timing behavior of timing events and timing event chains. Each event refers to a location of the AUTOSAR model where its occurrence is observed. An event chain is characterized by two events defining its beginning (stimulus) and end (response). Timing constraints are related to events or event chains. They define timing requirements which must be fulfilled by the system or timing guarantees that developers ensure regarding system behavior.

At Virtual Function Bus (VFB) level communication between SWCs is modeled by connected ports. We apply the Sender-Receiver paradigm in implicit mode, i.e. data elements are automatically read by the RTE before a Runnable is invoked and (different) data elements are automatically written after a Runnable has terminated [20]. AUTOSAR distinguishes *Inter-ECU* communication between two or more ECUs and *Intra-ECU* communication between Runnables on the same ECU [21]. For Inter-ECU communication, AUTOSAR supports the FlexRay protocol providing message transport in deterministic time slots [22]. FlexRay makes use of recurring communication cycles and is composed of a static and an optional dynamic segment. In the time-triggered static segment, a fixed and initially defined number of equally sized slots is statically assigned to one sender node. Changing this assignment requires a bus restart. Slot and frame size, cycle length, and several other parameters are defined by an initial setup of the FlexRay schedule. The payload segment of a FlexRay frame contains data in up to 127 2-byte words. Payload data can be divided into AUTOSAR *Protocol Data Units* (PDUs) composed of one or more words. Hence, different messages from one sender ECU can be combined by *frame packing*.

4 A Reconfigurable ECU Network Topology

To increase fault tolerance in an ECU network, node failures should be compensated by redundancy and software replication. In current distributed real-time



(a) Traction Control (b) Adaptive Cruise Control

Fig. 2. Functional Components of a TC (a) and an ACC (b) system [11]

systems, failures of hardwired nodes cannot be compensated by software redundancy as connections to sensors and actuators get lost. We propose a modified network topology distinguishing two types of ECUs [5]: (i) *Peripheral interface nodes*, which are wired to sensors and actuators, and just read/write values from/to the bus and (ii) *functional nodes* hosting the functional software and communicating over the bus. Since peripheral interface nodes do not execute complex tasks, they only require low hardware capacities allowing cost-efficient hardware redundancy. Here, we focus on distributed functional ECUs that provide and receive data via communication bus and can therefore be utilized for redundancy and reconfiguration. In the following ECU refers to functional nodes.

5 Fault-Tolerant Deployment Approach

In this section we present our fault-tolerant deployment approach. It contains (i) initial definition and modeling of the given SW-architecture & HW-topology for interdependent (ii) Runnable & task mappings and (iii) bus mappings. For better traceability all steps of our approach are applied to a real-world application.

5.1 Modeling of Software Architecture

Figure 2 illustrates the functional components of a Traction Control (TC) and an Adaptive Cruise Control (ACC) system, shows data dependencies, and provides information about their timing properties [11]: worst-case execution times (WCETs) and periods. In AUTOSAR these components are modeled as SWCs, whose functional behavior is represented by Runnables. Putting each Runnable into a separate SWC enables mapping of each Runnable to an arbitrary ECU. Thus, we use Runnable-to-ECU and SWC-to-ECU mapping as synonyms. The set of Runnables is modeled as $\mathcal{R} = \{\mathrm{R}_i(T_i, C_i, r_i, d_i, s_i, f_i) \,|\, 1 \leq i \leq n\}$. Each Runnable R_i is described by its period T_i, WCET C_i, release time r_i, deadline d_i, start time s_i, and finishing time f_i. For the TC system, Fig. 3 shows the

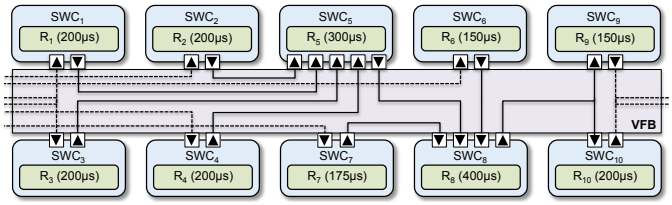


Fig. 3. Traction Control system model on AUTOSAR VFB level

resulting model on VFB level with Runnables listed in Table 1. A VFB level model represents the given software architecture with communication dependencies independent of the given hardware architecture and acts as input for the AUTOSAR-based system design. AUTOSAR Timing Extensions are used to annotate timing constraints for the model by means of events and event chains. Based on the event chain $R_1 \rightarrow R_5 \rightarrow R_8 \rightarrow R_{10}$ in Fig. 3, a maximum latency requirement defines that the delay between the input at R_1 (stimulus) and the output of R_{10} (response) must not exceed the given maximum end-to-end delay (period) of $3000\mu s$. Timing constraints are defined for each event chain. Dependencies between Runnables imply order and precedence constraints. Considering the maximum end-to-end delay for the event chains, we define an available execution interval $E_i = [r_i, d_i]$ for each Runnable R_i with release time:

$$r_i = \begin{cases} 0 & \text{, if } R_i \in \mathcal{R}_{\text{in}} \\ \max\{r_j + C_j \,|\, R_j \in \mathcal{R}_{\text{directPre}_i}\} & \text{, else.} \end{cases}$$

If a Runnable has no predecessors ($R_i \in \mathcal{R}_{\text{in}}$), the available execution interval of R_i starts at $r_i = 0$. Otherwise, r_i is calculated by means of r_j and C_j of the direct predecessors ($R_j \in \mathcal{R}_{\text{directPre}_i}$). The deadline of R_i is calculated as:

$$d_i = \begin{cases} \text{max end-to-end delay} & \text{, if } R_i \in \mathcal{R}_{\text{out}} \\ \min\{d_k - C_k \,|\, R_k \in \mathcal{R}_{\text{directSucc}_i}\} & \text{, else.} \end{cases}$$

If a Runnable has no successors ($R_i \in \mathcal{R}_{\text{out}}$), the available execution interval of R_i ends at $d_i =$ max end-to-end delay. Otherwise, d_i depends on r_k and C_k of the direct successors of R_i ($R_k \in \mathcal{R}_{\text{directSucc}_i}$). Table 1 summarizes the calculated available execution intervals E_i for the Runnables of the TC and ACC systems.

	TC System							ACC System							
R_i	R_{1-4}	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}
C_i	200	300	150	175	400	150	200	300	150	300	175	200	250	200	150
r_i	0	200	0	0	500	900	900	0	0	300	0	600	600	850	800
d_i	2100	2400	2400	2400	2800	3000	3000	2350	2350	2650	2550	2850	2800	3000	3000

Table 1. Runnable properties for TC and ACC systems (values in μs)

Algorithm 1 InitialMapping($\mathcal{R}, \mathcal{E}$)

Input: Runnables $\mathcal{R}$ and ECUs $\mathcal{E}$.
Output: RunnableMapping: $\mathrm{M_{init}}: \mathcal{R} \mapsto \mathcal{E}$.

```
1: SortByDeadlineAndReleaseTime(R)
2: for all R_i ∈ R_in do
3:    E_tmp ← GetFeasibleECUs(E)
4:    ECU_tmp ← GetEarliestFinish(E_tmp)
5:    MapRunnable(R_i, ECU_tmp)
6: end for
7: for all R_i ∈ R \ R_in do
8:    E_tmp ← GetFeasibleECUs(E)
9:    ECU_tmp ← GetMinimumDelay(R_i, E_tmp)
10:   MapRunnable(R_i, ECU_tmp)
11: end for
12: return M_init : R ↦ E
```

5.2 Runnable and Task Mapping

For a feasible SWC-to-ECU and Runnable-to-task mapping the properties of the ECUs have to be considered. The set of ECUs is $\mathcal{E} = \{\mathrm{ECU}_j \mid 1 \leq j \leq m\}$. We consider a homogeneous network structure. Hence, the Runnable WCETs provided in Table 1 are valid for all ECUs. The objective of our approach is to determine a feasible combined solution for an initial software deployment and all necessary reconfigurations and task replications for the remaining nodes of the network in case of a node failure. Thus, each configuration has to fulfill the deadlines of all Runnables and the end-to-end delay constraints for all event chains. Therefore, our approach iteratively analyzes and reduces the resulting execution delay for each SWC-to-ECU mapping to finally ensure minimized end-to-end delays for all event chains. It starts with the initial mapping $\mathrm{M_{init}}$ described by peusdo-code in Alg. 1. The algorithm defines the mapping order of Runnables via sorting them by deadline and release time. Before each mapping a schedulability test has to determine the feasible ECUs in $\mathcal{E}$. Therefore, we propose our *Extended Response Time Analysis* for Rate (Deadline) Monotonic Scheduling which is a common approach for AUTOSAR OS:

$$X_i = (C_i + \delta_i) + \sum_{j=1}^{i-1} \left\lceil \frac{X_i}{T_j} \right\rceil C_j.$$

It combines the WCET C_i and the resulting communication delay δ_i to calculate the response time X_i for each Runnable R_i. The initial mapping begins with the Runnables $\mathcal{R}_{\mathrm{in}}$, which have no precedence constraints, and maps them iteratively to the ECU hosting the last Runnable with the earliest finishing time. Thus, in a network with n ECUs, the first n Runnables will be mapped to empty ECUs. For Runnables with predecessors ($\mathrm{R} \in \mathcal{R} \setminus \mathcal{R}_{\mathrm{in}}$), Alg. 2 returns the ECU with the minimum execution delay. It determines the direct predecessors of R, their hosting ECUs $\mathcal{E}_{\mathrm{pre}}$, and the last Runnables on these ECUs ($\mathcal{R}_{\mathrm{last}}$). If

Algorithm 2 GetMinimumDelay(R, $\mathcal{E}$)

Input: A Runnable R and ECUs $\mathcal{E}$.
Output: ECU with minimum delay.

```
1: R_directPre ← GetDirectPredecessors(R)
2: E_pre ← GetHostECUs(R_directPre, E)
3: R_last ← GetLastRunnables(E_pre)
4: R_cap ← R_last ∩ R_directPre
5: if R_cap ≠ ∅ then
6:    ECU ← GetHostECU(GetLatestFinish(R_cap),E)
7: else
8:    ECU_preMin ← GetEarliestFinish(E_pre)
9:    if E \ E_pre ≠ ∅ then
10:      ECU_nonPreMin ← GetEarliestFinish(E \ E_pre)
11:      Δ ← Diff⟨GetFinishTime(ECU_preMin), GetFinishTime(ECU_nonPreMin)⟩
12:      if Δ > ComOverhead then
13:         ECU ← ECU_nonPreMin
14:      else
15:         ECU ← ECU_preMin
16:      end if
17:   else
18:      ECU ← ECU_preMin
19:   end if
20: end if
21: return ECU
```

one or more of the direct predecessors of R are last Runnable(s), the algorithm maps R to the same ECU as the predecessor with the latest finishing time. This avoids additional Inter-ECU communication delay for the latest input of R. If there are Runnables mapped to $\mathcal{E}_{\text{pre}}$ after all direct predecessors, the Inter-ECU communication for input to R can take place during their execution. In this case the algorithm determines the $\text{ECU}_{\text{preMin}}$ with the earliest finishing time. If there are ECUs that do not host any of the direct predecessors of R, the one with the earliest finishing time ($\text{ECU}_{\text{nonPreMin}}$) is also considered. The algorithm compares the difference Δ between these finishing times to the communication overhead resulting by a mapping to $\text{ECU}_{\text{nonPreMin}}$. The communication overhead depends on the number of slots needed and on the slot size defined for bus communication (ref. Section 5.3). If the communication overhead is smaller than Δ, the algorithm returns $\text{ECU}_{\text{nonPreMin}}$, else it returns $\text{ECU}_{\text{preMin}}$. By means of Alg. 1 and Alg. 2 our approach determines a feasible initial mapping with minimized execution delays considering timing, order, and precedence constraints. In a network with n ECUs it has to perform n redundancy mappings. Alg. 3 calculates the redundancy mapping M_{red} for a Runnable set to feasible remaining ECUs. Beside $\mathcal{R}_{\text{fail}}$ and $\mathcal{E}_{\text{rem}}$ it takes M_{init} as an input, meaning that the set of Runnables initially mapped to $\mathcal{E}_{\text{rem}}$ is kept for each remaining ECU. This allows to combine Runnables on $\mathcal{E}_{\text{rem}}$ to tasks and the reuse of messages and slots in different reconfigurations. Similar to the initial mapping, the algorithm

Algorithm 3 RedundancyMapping($\mathcal{R}_{\text{fail}}, \mathcal{E}_{\text{rem}}, \text{M}_{\text{init}}$)

Input: Runnables $\mathcal{R}_{\text{fail}}$, ECUs $\mathcal{E}_{\text{rem}}$, and Mapping M_{init}.
Output: RedundancyMapping: $\text{M}_{\text{red}} : \mathcal{R}_{\text{fail}} \mapsto \mathcal{E}_{\text{rem}}$.

```
1: M_red ← M_init
2: for all R_i ∈ R_fail do
3:     E_tmp ← GetFeasibleECUs(E_rem)
4:     ECU_tmp ← GetECUMinE2E(R_i, E_tmp, M_red)
5:     M_red ← M_red ∪ MapRunnable(R_i, ECU_tmp)
6: end for
7: return M_red : R_fail ↦ E_rem
```

iteratively inserts the Runnables from $\mathcal{R}_{\text{fail}}$. Hence, in each mapping step the redundancy mapping M_{red} is complemented by the currently performed mapping. In Alg. 4, for each assignment our approach determines the Runnable-to-ECU mapping resulting in the minimum overall end-to-end delay, i.e. longest end-to-end delay of all event chains. This algorithm checks each $\text{ECU}_i \in \mathcal{E}_{\text{rem}}$ based on their current mapping. It complements M_{cur} by inserting R preserving order and precedence constraints by means of deadlines and release times. This insertion results in Runnable shiftings and growing execution delays due to the constraints on one or more of the ECUs. The algorithm calculates the overall end-to-end delay for all event chains implied by M_i and stores it referencing to ECU_i. This results in a set of end-to-end delays (E2E): one for each Runnable-to-ECU mapping. Finally, Alg. 4 compares these values and returns the ECU with minimum overall end-to-end delay. Fig. 4 depicts Gantt Charts for the TC and ACC systems in a network with 3 ECUs. It shows how our SWC-to-ECU approach preserves the initial order of Runnables on remaining ECUs and inserts redundant Runnables. It also shows that our approach enables an efficient Runnable-to-task mapping to reduce the number of required tasks. For this purpose Runnables that are assigned to the same ECU and keep connected at each redundancy mapping, are encapsulated in one task. Summarized, this results in 13 tasks for the initial mapping and 18 tasks for the redundant Runnables. Although each redundant Runnable is mapped to a separate task, our approach also supports the encapsulation of redundant Runnables in one task.

Algorithm 4 GetECUMinE2E($\text{R}, \mathcal{E}, \text{M}_{cur}$)

Input: Runnable R, ECUs $\mathcal{E}$, and Mapping M_{cur}.
Output: ECU causing minimum overall E2E delay

```
1: for all ECU_i ∈ E do
2:     M_i ← M_cur ∪ MapRunnable(R, ECU_i)
3:     E2E_ECU_i ← OverallE2EDelay(M_i)
4:     E2E ← E2E ∪ E2E_ECU_i
5: end for
6: ECU ← ECUMinE2E(E2E)
7: return ECU
```

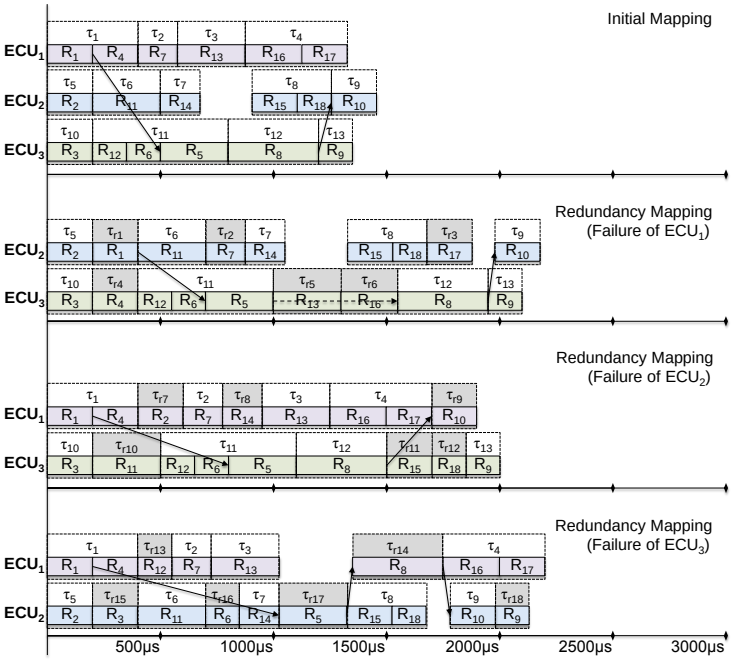


Fig. 4. Gantt Charts of Runnable and task mappings for TC and ACC systems

5.3 Communication and Bus Mapping

The number of Inter-ECU messages depends on the Runnable mappings; their size depends on the given software architecture. The message sizes of the TC and ACC systems are 10 to 22 *bits* [11] and require one or two words of a FlexRay frame. For each Inter-ECU message m_i, the Runnable mappings result in an available transmission interval $\mathrm{Tx}_i = [f_{\mathrm{send}}, s_{\mathrm{recv}}]$. Thus, m_i may be transmitted in one slot θ of the slot set Θ_i in Tx_i. The number of slots in Θ_i depends on the slot size. Here, we consider a slot size of $\theta_{\mathrm{size}} = 25\mu s$, i.e. up to 6 PDUs per frame. Alg. 5 describes our bus mapping approach. It adds the Inter-ECU messages $\mathcal{M}_{\mathrm{M}_i}$ of all Runnable mappings to a common message set $\mathcal{M}$. For each message it determines the sender ECU and transmission interval per mapping and adds it to a common set Ω_{m_j}. Afterwards, it performs an assignment of slots to messages respectively sender ECUs. Therefore, all Inter-ECU messages $\mathcal{M}$ are considered. Analyzing all messages with the same sender ECU, the corresponding transmission intervals $\Omega_{\mathrm{m}_i,\mathrm{ECU}_j}$ get identified. Since the initial mapping is kept, Inter-ECU messages can be sent by the same ECU in one or more Runnable mappings (ref. Fig. 4). Thus, our approach reduces the number of needed slots for the ECU assignments. It compares the determined transmission intervals. For overlapping slots the first available common slot θ_{map} is assigned to the sender ECU for the transmission of m_i. Thus, the same message and slot is reused in

Algorithm 5 BusMapping($\Pi, \Theta, \mathcal{E}$)

Input: RunnableMappings Π, Set of Slots Θ, and ECUs $\mathcal{E}$.
Output: BusMapping: $(\mathcal{M}, \mathcal{E}) \mapsto \Theta$

for all $\mathrm{M}_i \in \Pi$ **do**
  $\mathcal{M}_{\mathrm{M}_i} \leftarrow$ GetInterEcuMSGs(M_i)
  $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{M}_{\mathrm{M}_i}$
  for all $\mathrm{m}_j \in \mathcal{M}_{\mathrm{M}_i}$ **do**
    $\mathrm{ECU}_{\mathrm{send}(\mathrm{M}_i,\mathrm{m}_j)} \leftarrow$ GetSenderECU(M_i, m_j,$\mathcal{E}$)
    $\Theta_{\mathrm{m}_j,\mathrm{ECU}_{\mathrm{send}(\mathrm{M}_i,\mathrm{m}_j)}} \leftarrow$ GetTxInterval (m_j, $\mathrm{ECU}_{\mathrm{send}(\mathrm{M}_i,\mathrm{m}_j)}$)
    $\Omega_{\mathrm{m}_j} \leftarrow \Omega_{\mathrm{m}_j} \cup \Theta_{\mathrm{m}_j,\mathrm{ECU}_{\mathrm{send}(\mathrm{M}_i,\mathrm{m}_j)}}$
  end for
end for
for all $\mathrm{m}_i \in \mathcal{M}$ **do**
  $\mathcal{E}_{\mathrm{sender}} \leftarrow$ GetSenderECUs(Ω_{m_i},$\mathcal{E}$)
  for all $\mathrm{ECU}_j \in \mathcal{E}_{\mathrm{sender}}$ **do**
    $\Omega_{\mathrm{m}_i,\mathrm{ECU}_j} \leftarrow$ GetTxIntervalsForSameSender(Ω_{m_i}, ECU_j)
    $\theta_{\mathrm{map}} \leftarrow$ GetFirstCommonAndAvailableSlot($\Omega_{\mathrm{m}_i,\mathrm{ECU}_j}$)
    MapToSlot((m_i, $\mathrm{ECU_j}$),θ_{map})
  end for
end for
return $(\mathcal{M}, \mathcal{E}) \mapsto \Theta$

Message	Sender: Transmission Interval (μs)	Slot (μs)
$\mathrm{R_1} \rightarrow \mathrm{R_5}$	$\mathrm{ECU_1}$:[200, 500] , [200, 800] , [200, 1025]	[200, 225]
	$\mathrm{ECU_2}$:[400, 700]	[400, 425]
$\mathrm{R_2} \rightarrow \mathrm{R_5}$	$\mathrm{ECU_1}$:[600, 800]	[600, 625]
	$\mathrm{ECU_2}$:[200, 500] , [200, 700]	[225, 250]
$\mathrm{R_4} \rightarrow \mathrm{R_5}$	$\mathrm{ECU_1}$:[400, 500] , [400, 800] , [400, 1025]	[425, 450]
$\mathrm{R_5} \rightarrow \mathrm{R_8}$	$\mathrm{ECU_2}$:[1325, 1350]	[1325, 1350]
$\mathrm{R_6} \rightarrow \mathrm{R_8}$	$\mathrm{ECU_2}$:[850, 1350]	[1325, 1350]

Table 2. Bus Mapping for Inter-ECU communication (excerpt)

different reconfigurations. In non-overlapping intervals, m_i is mapped to the first available slot in each interval. It is also checked if the current message can be mapped to the same slot as one of the other messages by utilizing frame packing. Table 2 provides an excerpt of the determined bus mappings. It shows transmission intervals and assigned slots for messages per sender and gives examples for reuse and frame packing. Fig. 4 depicts the end-to-end delays for the event chain $\mathrm{R_1} \rightarrow \mathrm{R_{10}}$ and shows that the end-to-end delay constraint is fulfilled for all mappings. The same holds for all other event chains.

6 Reconfiguration with AUTOSAR

Having a feasible AUTOSAR-compliant SWC-to-ECU and Runnable-to-task mapping, two challenges remain to solve by means of AUTOSAR: Detect a failed

ECU and activate the appropriate redundant tasks within the ECU network according to the fault-tolerant reconfiguration. While AUTOSAR specifies a BSW called Watchdog Manager to manage errors of BSW modules and SWCs running on an ECU, there is no explicit specification regarding detection of failed nodes within an ECU network. Therefore, we propose to extend AUTOSAR BSW by means of a Complex Device Driver (CDD, [23]). Using FlexRay-specific functionality provided by BSW of AUTOSAR Communication Stack, it can be monitored if valid frames are received. Combined with the static slot-to-sender assignment, each ECU can identify failed ECUs. When a failed ECU is detected, each remaining ECU has to activate its appropriate redundant tasks. For this purpose we propose using *ScheduleTables*: a statically defined activation mechanism provided by AUTOSAR OS for time-triggered tasks used with an OSEK Counter [13]. Here, we use the FlexRay clock to support synchronization of ScheduleTables running on different ECUs within a network. Note that tasks are only activated, i.e. tasks require an appropriate priority to ensure that they are scheduled in time. For each ECU we define one single ScheduleTable for each configuration of this ECU, i.e. a ScheduleTable activates only those tasks that are part of its corresponding configuration. Utilizing the different states that each ScheduleTable can enter – e.g. *RUNNING* and *STOPPED* – the ScheduleTable with the currently required configuration is RUNNING while all the others are STOPPED. Since in this paper we consider periodic tasks, ScheduleTables have repeating behavior, i.e. a RUNNING ScheduleTable is processed in a loop. Having an AUTOSAR-compliant concept to detect a failed ECU within a network and to manage different task activation patterns on an ECU, we need to combine these concepts. This can be done by using BSW Mode Manager. Defining one mode per configuration on a particular ECU, our CDD can request a mode switch when a failed ECU is detected. This mode switch enforces that the currently running ScheduleTable is STOPPED and, depending on the failed ECU, the appropriate ScheduleTable enters state RUNNING.

7 Conclusion

We presented an approach for fault-tolerant deployment of real-time software in AUTOSAR ECU networks and applied it to real-world applications. It offers methods for task and message mappings to determine an initial deployment combined with reconfigurations. To enable redundancy, we proposed a reconfigurable network topology. Finally, we introduced a CDD for detecting failed nodes and activation of reconfigurations.

8 Acknowledgements

This work was partly funded by the DFG SFB 614 and the German Ministry of Education and Research (BMBF) through the project SANITAS (01M3088I) and the ITEA2 projects VERDE (01S09012H), AMALTHEA (01IS11020J), and TIMMO-2-USE (01IS10034A).

References

1. Fennel, H., et al.: Achievements and exploitation of the AUTOSAR development partnership. In: Society of Automotive Engineers (SAE) Convergence. (2006)
2. Scheickl, O., Rudorfer, M.: Automotive real time development using a timing-augmented autosar specification. In: Proceedings of the 4th European Congress on Embedded Real-Time Software (ERTS). (2008)
3. Burns, A.: Scheduling hard real-time systems: A review (1991)
4. Paret, D.: Multiplexed Networks for Embedded Systems. Wiley (2007)
5. Klobedanz, K., et al.: An approach for self-reconfiguring and fault-tolerant distributed real-time systems. In: 3rd IEEE Workshop on Self-Organizing Real-Time Systems (SORT). (2012)
6. Pop, P., et al.: Scheduling with optimized communication for time-triggered embedded systems. In: Proceedings of the 7th international workshop on Hardware/software codesign (CODES). (1999)
7. Pop, P., et al.: Bus access optimization for distributed embedded systems based on schedulability analysis. In: Proceedings of Design, Automation and Test in Europe (DATE). (2000)
8. Eles, P., et al.: Scheduling with bus access optimization for distributed embedded systems. IEEE Trans. Very Large Scale Integr. Syst. **8**(5) (2000)
9. Ding, S., et al.: A ga-based scheduling method for flexray systems. In: Proceedings of the 5th ACM international conference on Embedded software (EMSOFT). (2005)
10. Ding, S., et al.: An effective ga-based scheduling algorithm for flexray systems. IEICE - Transactions on Information and Systems **E91-D**(8) (2008)
11. Kandasamy, N., et al.: Dependable communication synthesis for distributed embedded systems. In: International Conference on Computer Safety, Reliability and Security (SAFECOMP). (2003)
12. Brendle, R., et al.: Dynamic reconfiguration of flexray schedules for response time reduction in asynchronous fault-tolerant networks. In: Architecture of Computing Systems (ARCS). (2008)
13. AUTOSAR: Specification of Operating System, Ver. 5.0.0 (2011)
14. Peng, W., et al.: Deployment optimization for autosar system configuration. In: 2nd International Conference on Computer Engineering and Technology (ICCET). (2010)
15. Zhang, M., Gu, Z.: Optimization issues in mapping autosar components to distributed multithreaded implementations. In: 22nd IEEE International Symposium on Rapid System Prototyping (RSP). (2011)
16. Kim, J., et al.: An autosar-compliant automotive platform for meeting reliability and timing constraints. In: Society of Automotive Engineers (SAE) World Congress and Exhibition. (2011)
17. AUTOSAR: Software Component Template, Ver. 4.2.0 (2011)
18. AUTOSAR: Specification of Timing Extensions, Ver. 1.2.0 (2011)
19. Peraldi-Frati, M.A., et al.: Timing modeling with autosar - current state and future directions. In: Design, Automation, and Test in Europe Conference Exhibition (DATE). (2012)
20. AUTOSAR: Specification of RTE, Ver. 3.2.0 (2011)
21. AUTOSAR: Specification of Communication, Ver. 4.2.0 (2011)
22. FlexRay Consortium: FlexRay Communications System Protocol Specification Ver. 2.1. (2005)
23. AUTOSAR: Technical Overview, Ver. 2.1.1 (2008)