Reconfigurable Communication Middleware for FlexRay-based Distributed Embedded Systems

Diptesh Majumdar*, Licong Zhang†, Purandar Bhaduri*, and Samarjit Chakraborty†
*Department of Computer Science and Engineering, IIT Guwahati, India
{diptesh, pbhaduri}@iitg.ernet.in
†Institute for Real-Time Computer Systems, TU Munich, Germany
{licong.zhang, samarjit}@tum.de

***Abstract*—In this paper we consider the case of a network of Electronic Control Units (ECUs) connected through a FlexRay bus in the automotive domain. Multiple distributed applications can run on this underlying architecture, each partitioned into tasks that are mapped on different ECUs. These applications can often be executed in different functional modes with different requirements on the communication resources in terms of data size and sampling period. Moreover, new applications can be deployed on to the ECUs at run-time. To efficiently utilize the communication resources and accommodate new applications, a certain flexibility in reallocation of the resource is necessary. However, the FlexRay bus requires static configuration of schedules and data mapping in order to guarantee a more deterministic system behavior, allowing little room for flexibility. In order to address this problem, we propose a reconfigurable communication middleware that lies between the application layer and the communication controller layer, which maps messages onto FlexRay schedules, and can be reconfigured at runtime. The configuration is synthesized and deployed online, allowing a certain reallocation of communication resources to applications. In this paper, we describe the design of such a reconfigurable communication middleware and demonstrate its function with an implementation using industry-strength FlexRay design tools.**

I. Introduction

In modern vehicles, increasingly more complicated software applications are developed to help control the vehicle, assist the driver and offer more comfortable driving experience. This development has led to an increase in the scale and complexity of the Electrical/Electronic architecture, and also imposes more load on the existing communication buses. As the communication resources have become more scarce and valuable, an efficient utilization of the bandwidth has become a problem of great interest. Furthermore an application can sometimes run in different operation modes, offering different levels of performance. Each mode may be characterized by a different algorithm or sampling period of tasks, data size, period of the messages sent over communication bus, etc.. An example of this is a control application (e.g., engine control), where different environment conditions or driving patterns might require different modes. If a vehicle runs in a difficult terrain like paths on a mountain, it requires a better performance compared to driving on a city street with moderate speed, thus requiring better control and more data sent on the in-vehicle network. Another example can be an object detection and collision warning driver assistance system. If the vehicle is driving at lower speed, the safe distance could be shorter, in which case less data needs to be processed and communicated. A safe design paradigm for this case is over-provisioning of resources for the applications so that they always have enough resources to run in the most demanding mode. However, there can be a waste of valuable computation and communication resources whenever the applications are not required to run in the best possible mode. Moreover, it is difficult in this case to accommodate Plug-and-Play applications, i.e., new applications deployed onto the vehicle while it is in use, which is an emerging trend in the automotive domain. In addition, in the automotive context, the case of inadequate communication resources is more likely to happen, due to the constantly increasing number of applications and size of data. In this case, a likely scenario is that not all applications can run simultaneously in their best mode. Thus the ability of the system to allow resource sharing so that the applications can switch between different operation modes at run time is quite important. However, to enable this, a certain flexibility of the system is necessary. Furthermore, the support for future Plug-and-Play ability of the vehicles also requires some flexibility and scalability of the underlying embedded system. But the design and implementation of current automotive bus systems does not allow much flexibility. Let us consider FlexRay for example, which is a typical automotive communication bus system usually used for the safety critical applications. The FlexRay clusters are often designed and configured offline and there is little room for an online reconfiguration of the system on the protocol level.

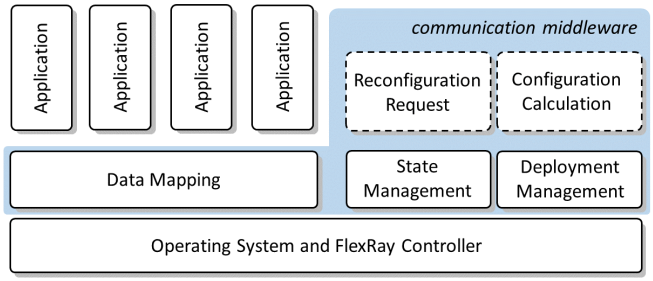

Fig. 1: The proposed communication middleware.

Towards addressing this problem, in this paper we propose a communication middleware for FlexRay-based distributed embedded systems, enabling a certain reallocation of the communication resources within an ECU. In this middleware, we introduce a data mapping component that casts messages of applications into FlexRay frames with a specific schedule based on a configuration. We use additional software components to synthesize the configuration of both the application mode and the data mapping in order to deploy the new configuration and to safely reconfigure the system. This communication middleware will enable a reconfiguration of the communication on a FlexRay bus and facilitate the online change of application modes and activation of newly deployed

applications. The experimental results have shown that the proposed middleware can be implemented in a FlexRay based ECU system using a commercial off-the-shelf tool-chain.

Related work: Mundhenk *et al.* [1] have introduced a virtual event-triggered communication layer on top of a time-triggered communication infrastructure to flexibly schedule policy-based messages. At run-time, event-triggered messages with assigned priorities are sorted in the virtual layer to produce wrapper Protocol Data Units (PDUs) that fill the entire static slot. However, the message mapping is based on a pre-defined policy and the change in the period of messages according to change in mode of applications has not been considered. For the multi-mode applications, Phan *et al.* [2] presented a Mode Change Protocol (MCP) to express the system behavior during mode transitions in multi-mode real time systems. Each MCP model is a finite state automaton, represented as a DAG where each node captures the system tasks during a mode transition and each edge specifies a buffer update and a timing/buffer condition between two intermediate transitional stages. Sha *et al.* [3] proposed a mode change protocol supporting mode changes in the context of common preemptive scheduling algorithms for periodic tasks. A comprehensive survey of mode change protocols for real-time systems can be found in [4]. Also, the construction of multi-mode schedules by extension of a single mode scheduler has been demonstrated in [5]. However, these works do not address the problem of FlexRay communication, while in this work, we focus on the scheduling at the communication level instead. A method to reconfigure a FlexRay network in order to increase fault tolerance has been proposed in [6]. This approach is mainly targeted at the event of a node failure since it uses redundant slots in the schedule which again under-utilizes the bandwidth of communication in normal circumstances, a condition we aim to improve in the present work.

Our contribution: In this paper we demonstrate a method that allows reconfiguration of applications and allocation of communication resources within an ECU. In order to perform this reconfiguration, we propose a communication middleware layer between the application layer and the operating system and communication controller layer. This middleware layer allows an online calculation of configuration and deployment of the new configuration on a reconfiguration request. The calculation involves generation of a feasible mode combination of the currently running applications along with new schedules for periodic messages while maximizing the overall system performance level. Based on the newly generated configuration, the middleware allows re-mapping of messages onto underlying static communication resources.

The rest of this paper is organized as follows. In Section II we present the problem formulation including the architectural setting we are considering including the FlexRay communication protocol. The proposed communication middleware layer, its components and their function are explained in Section III. Section IV shows the experimental results, where a case study is used to demonstrate the online reconfiguration of applications of the proposed middleware using industry-strength FlexRay desgin tools, before we conclude in Section V.

II. Problem Formulation

A. Architectural Setting

In this paper we consider a distributed embedded system consisting of multiple ECUs connected by a common FlexRay bus. We denote the set of ECUs as $\mathcal{E} = \{E_1, ..., E_{Ne}\}$, where Ne is the number of ECUs in the system. We assume that a set of *applications* are running in the system, denoted by $\mathcal{A} = \{a_1, ..., a_{Na}\}$. An application a_i is a collection of *tasks* and *messages* that performs an independent function. The tasks in an application can be mapped on different ECUs and the data between them are sent on the FlexRay bus as messages.

We denote a task as τ_j and the set of tasks belonging to an application a_i as $\mathcal{T}_i$. A message contains the actual data that needs to be sent on the communication medium. It can be characterized by a tuple $m_j = (w_j, p_j, \Theta_j)$, where w_j and p_j denote respectively the data size per message and the period at which the message needs to be sent. Θ_j represent the FlexRay schedule for the message, which will be explained later. Here we differentiate between a message and a frame. A message is a certain amount of raw data that needs to be packed into a FlexRay frame to be transmitted on the bus.

An application can have different pre-programmed operation *modes*. Each mode is a customization of the application, offering a different level of functional performance. Depending on the performance to provide, it could require different amount of data to be sent on the bus. This variation could lie in the data size or the period of the message. Furthermore, the tasks of the application might also have a different period or algorithm implementation for a different mode. An application mode is also associated with a specific *performance value*, where usually a higher performance value requires more resources i.e. more data size per message and (or) greater sending frequency. This performance value is usually specified by the application designer and can possibly be derived from the characteristics of the tasks and messages. Here we characterize an application as $a_i(\alpha_i) = (\mathcal{T}_i(\alpha_i), \mathcal{M}_i(\alpha_i), J_i(\alpha_i))$, where $\alpha_i \in \mathcal{G}_i$ denotes the mode and $\mathcal{G}_i$ is the set of pre-defined operation modes of the application a_i. $\mathcal{T}_i$ and $\mathcal{M}_i$ denote respectively the set of all the tasks and messages in the application. $J_i \in \mathcal{J}_i$ is the corresponding performance value of mode α_i. We can reasonably assume that all characteristics of $\mathcal{T}_i$ and $\mathcal{M}_i$ as well as $\mathcal{J}_i$ for all modes are known beforehand, which is provided by the application designer and stored in the *manifest* of the application in the ECUs that the application is mapped on. In terms of $\mathcal{M}_i$, the corresponding requirements of data size and period should be specified.

B. FlexRay Communication Protocol

FlexRay [7] is a communication protocol commonly found in the automotive domain and is suitable for safety critical application clusters. As a hybrid protocol, it offers both *time-triggered* and *event-triggered* communication services. FlexRay is organized as a series of periodic *communication cycles*, which we denote here as T_{bus}. Each cycle has two major components, the *static segment* (ST) and the *dynamic segment* (DYN), where the time-triggered and event-triggered mechanisms are respectively applied. Every 64 communication cycles constitute a periodic sequence of bus cycles, as shown in Fig. 2. The static segment of a communication cycle follows the TDMA approach, where the whole segment is partitioned into a number of *static slots* of equal length Δ. We denote the static slots in a communication cycle as $S_{ST} = \{1, ..., S_{Ls}\}$.

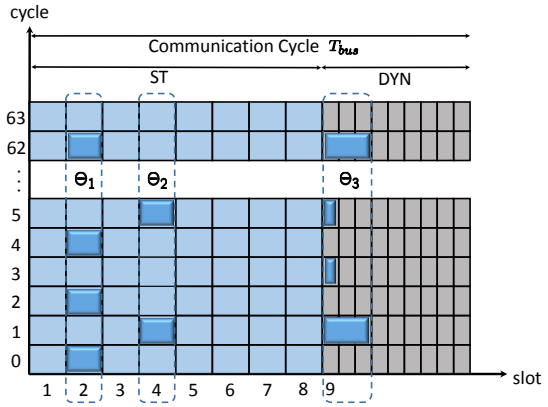

Fig. 2: An Example of FlexRay schedules: $\Theta_1 = (2, 0, 2)$, $\Theta_2 = (4, 1, 4)$, $\Theta_3 = (9, 1, 2)$.

A *frame* can be assigned to one static slot. It does not matter whether there is data to be sent or not in a communication cycle, the static slot will always be occupied. The dynamic segment employs the Flexible-TDMA (FTDMA) approach. In the context of this paper, we only consider the case where data are sent on the static segment.

Here we can divide the concept of a communication schedule for FlexRay into two different layers. Schedule on the communication layer, or protocol layer represents the actual timing when the FlexRay frame is sent on the bus, i.e., the slot where the frame is assigned to. Schedule on the application layer represents the mapping of messages onto frames. The schedule on the two layers together constitute the schedule of a message. Here we first define the FlexRay schedule on the communication layer. As explained above, FlexRay is organized as a periodic sequence of 64 communication cycles, with each communication cycle indexed by a *cycle counter*. The cycle counter increments from 0 to 63 and is then reset to 0. Therefore to fully characterize a FlexRay schedule on the communication layer, we can define a schedule as $\Theta_i = (S_i, B_i, R_i)$, where S_i represents the slot number, R_i represents the *repetition rate* and B_i represents the *base cycle*. The repetition rate is the number of communication cycles that has to elapse between two consecutive transmission of the frame and can take the value $R_i \in \{2^n | n \in \{0, ..., 6\}\}$. The base cycle indicates the cycle offset in the 64 cycles. Fig. 2 shows an example of FlexRay schedules. Multiple slots can be assigned to a specific ECU and additionally a slot can have multiple FlexRay schedules through *slot multiplexing*. On the application layer, a message needs to be packed into FlexRay frames, which is essentially a mapping of messages on FlexRay schedules.

C. Constrained Communication Resources

In an automotive setting, we often face a case of constrained communication resources, where a common bus is shared by many ECUs. This is especially true in the case of a FlexRay-based system, where during the incremental design a FlexRay cluster configuration is often inherited from previous designs while more ECUs and applications are mapped onto the system. Therefore, over-provisioning communication resources for applications is not an efficient design paradigm. In this paper, we consider the case that for a specific ECU E_i, a set of static slots $S_j \in \mathcal{S}(E_i)$ are already assigned. We try to reallocate the underlying static communication resources available to the applications at run time.

Note that the requirement for computation resource often also varies with application modes. For example, if an application switches to different modes, the task period, the execution time and schedules can also change. This problem of task scheduling in the case of multi-mode applications is fortunately well studied, as mentioned in the related work [2], [3], [4], [5]. In this paper, we assume the scheduling problem at the task-level can be solved by existing methods and focus on the scheduling at the communication level instead.

III. Proposed Communication Middleware

A. Motivational Example

Let us illustrate the problem with a concrete example. Consider the case of a subsystem consisting of two ECUs, E_1 and E_2, which is a part of a whole ECU network. There are currently applications a_1, a_2 and a_3 partitioned and mapped on both ECUs. Each application has one task mapped on E_1 and a following task mapped on E_2. Periodic messages m_1, m_2 and m_3 are sent from E_1 to E_2. The application manifest for messages is shown in Table I. Let us assume that the duration of one communication cycle of the FlexRay bus is T_{bus} and a static slot can accommodate a maximal payload of 8 bytes. Two static slots S_1 and S_2 are assigned to E_1.

We have a case of a constrained communication resources here, since the available static slots assigned can not accommodate all three applications to operate in their best mode, i.e., mode 1. Therefore, some of the applications can only run in modes with lower performance. Assume the overall performance value of the system is a weighted sum of the performance values of all applications with equal weights, Table II shows some combination of modes with better overall performance values. In the conventional design, one of these combinations is statically configured and the system can only run as such. For example, if we choose combination 1 here, then it is not possible for application a_3 to switch to mode 1 at run time. Even when the ECU allows the tasks of the applications to switch at runtime, no sufficient communication resources can be allocated. The only way is to over-provision the communication resources to accommodate all the applications in their best mode. But if it is not necessary for applications to run in the best mode, this will be a waste of valuable communication resources. Consider a second case, where three applications are running in the system and a fourth application a_4 is deployed on E_1 and E_2. The application can not be activated since there is no communication resources assigned to it and the application will not be able to transmit any data on the bus, even though it is theoretically possible to map the message of a_4 on top of combination 1. In this case, even over-provisioning does not help since it may not be possible to know at design time the manifest of the new application and the system will not be able to transmit the data due to lack of necessary configuration of the extra message.

To enable the mode switch of the applications and accommodation of messages of new applications on the FlexRay bus at runtime, we need more flexibility in the FlexRay communication to allow a certain degree of reallocation of communication resources at runtime. A clear interface to the applications is also important so that a newly developed application just needs a manifest for the FlexRay communication to reconfigure to accommodate the new messages.

a_1				a_2				a_3				a_4			
α_1	w_1	p_1	J_1	α_2	w_2	p_2	J_2	α_3	w_3	p_3	J_3	α_4	w_4	p_4	J_4
1	8	T_{bus}	100	1	4	$2T_{bus}$	100	1	8	T_{bus}	100	1	4	$2T_{bus}$	100
2	4	T_{bus}	80	2	4	$4T_{bus}$	80	2	4	T_{bus}	80	Off			0
3	4	$2T_{bus}$	50	3	4	$8T_{bus}$	50	3	4	$2T_{bus}$	50				
Off			0	Off			0	Off			0				

TABLE I: Communication manifest of the applications in the motivational example.

index	a_1	a_2	a_3	$J_{overall}$	index	a_1	a_2	a_3	$J_{overall}$
1	1	1	2	280	4	2	2	1	260
2	2	1	1	280	5	1	1	3	250
3	1	2	2	260	6	3	1	1	250

TABLE II: Combination of application modes with better performance levels.

B. Software Architecture

In this paper, we propose a communication middleware architecture that enables online reconfiguration of FlexRay communication. As shown in Fig. 1, the proposed middleware consists mainly of five software components: (i) *data mapping*, (ii) *state management*, (iii) *deployment management*, (iv) *re-configuration request* and (v) *configuration calculation*.

Before describing each software component, we first explain the concept of *configuration* considered in this paper. We divide the configuration of the system into two parts. The first part is the configuration of applications, which is essentially a list of active applications and their operating modes. This can be characterized by a set $C_a = \{\alpha_i | a_i \in \mathcal{A}\}$, where $\mathcal{A}$ is the set of applications considered. Based on α_i, application a_i can adjust, for example, the algorithms and the schedules of the tasks and the amount of data that needs to be transmitted. The second part is the configuration for data mapping, i.e., it decides which message is packed into a frame with a specific FlexRay schedule. We denote this configuration as $C_c = \{\mathcal{M}_i | a_i \in \mathcal{A}\}$. For each message $m_j \in \mathcal{M}_i$ involved, a mapping $m_j \rightarrow \Theta_j$ is obtained. The data size w_j and period p_j are obtained from the manifest and the mode of the applications. The configuration C_c can also be seen as a lookup table for data mapping.

The data mapping component is responsible for mapping messages of different applications to the available FlexRay slots. The state management takes care of the safe state transition of the whole system in the re-configuration process. The deployment management component deploys the new configuration to the relevant software components and manages the reconfiguration of the applications and the mapping component. The reconfiguration request component negotiates the reconfiguration request and sends it to the configuration calculation component, which calculates the suitable new configuration and passes it onto the deployment management module. The interfaces and interaction between the components are shown in Fig. 3. When there is no reconfiguration request, the applications and data mapping component work normally based on an available configuration.

C. Data Mapping

The data mapping component maps application data into the corresponding FlexRay slot according to the configuration C_c. On the sender side, it packs the messages into the slots and on the receiver side it retrieves the messages and passes them onto the corresponding application tasks. We consider a set of static slots $\mathcal{S}_i$ assigned to ECU E_i. For each $S_j \in \mathcal{S}_i$, there could be multiple FlexRay schedules

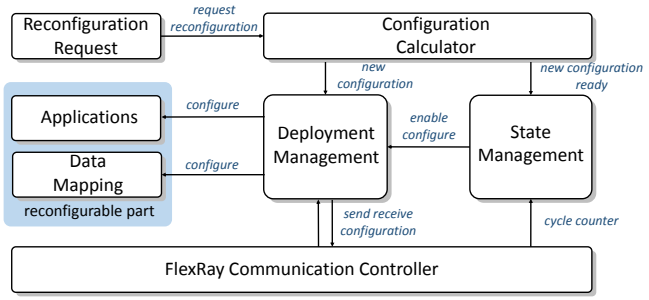

Fig. 3: Interface between the software components.

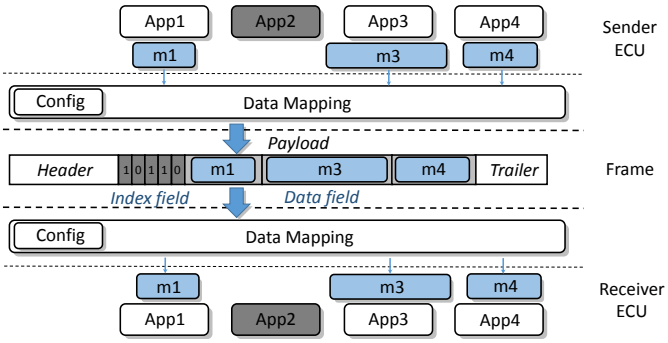

Fig. 4: Data mapping component and the message to frame mapping.

for the slot depending on the repetition rate and base cycle of the schedule. In the conventional design paradigm, the messages are statically mapped to frames, which has a FlexRay schedule. Therefore, it is not possible to reconfigure the system when new data or different data are sent. Towards tackling this problem, in the middleware proposed, we assign a base schedule $\Theta_{j,base} = (S_j, 0, 1)$ to each slot and assign a frame with maximum payload allowed for the slot, i.e., a longest possible frame is sent every communication cycle on the slot. The data mapping component is then responsible to cast the messages into the payload of the frames according to a mapping table, which can be reconfigured online. In this paper, we consider the case where every ECU is synchronized to the FlexRay controller and the FlexRay communication cycle counter needs to be known to the middleware.

As shown in Fig. 4, the application data are packed into and unpacked out of the frame by the data mapping component. The payload of a frame can be divided into two fields, the *index field* - which identifies the applications whose data are packed into the current frame - and the *data field* - which holds the actual messages. If the index field is used, the length of the index field depends on the maximal number of applications that can be considered. One bit is assigned to each application. In this case, no configuration C_c needs to be deployed for a receiver ECU since it can determine which message is packed into the frame based on the index field and manifest of the application. Alternatively, the configuration C_c can be deployed and stored on the receiver ECU, in which case

an index field is not necessary. The mapping component on the sending side packs the messages of applications into the frame according to C_c. It sets the index field (if applied) to indicate the application whose message is contained in this frame. Then it casts the messages into the data field of the payload, ordered according to the index of the applications. To keep the problem simple, if multiple messages are sent by an application, we put the messages together as a large single message. On the receiving side, it receives the frame and reads first the index field to know the messages of which applications have been sent on the frame. Then it cast the data field into the messages based on the application manifest and forward the them to the corresponding applications. Casting of the data field requires the knowledge of configuration C_a, which along with the manifest lets the receiver know of the expected width and type of the incoming data. Note that if the index field is applied, the configuration C_c is indicated through the index field and thus does not need to be deployed on the receiving side. Alternatively the configuration C_c can first be deployed on the receiving side and the data mapping component can then cast the payload into messages according to C_c. In that case, an index field is not necessary. Here we only described the sending and receiving side behavior of the data mapping component. If an ECU is simultaneously a sender and a receiver, the data mapping component will contain the function of both sides.

D. Configuration Deployment

The configuration deployment management component is responsible for the deployment of the configuration. Its function can be divided into two phases: (i) sending the configuration to all relevant ECUs and (ii) actual reconfiguration of the application tasks and the data mapping component of the middleware. For phase (i), we introduce an additional configuration deployment application a_{cd}, which is associated with a message m_{cd}, which will be broadcast from the ECU hosting the configuration calculator to all the relevant ECUs. This message contains C_a and C_c, which can be cast into a specific data type, possibly a bit stream. If the message is longer than the maximal payload allowed, it can be segmented and transmitted on multiple frames and a signal is introduced to specify the start and end of the configuration transmission. This message can be mapped on a reserved static slot with any valid repetition rate and can also share a slot with other messages using slot multiplexing. Once the ECUs receive the configuration C_a and C_c (if C_c is necessary), it is stored and the ECUs wait for the synchronized reconfiguration time point. Until then, the ECUs continue the communication according to the previous configuration. Once they reach the reconfiguration time point, i.e., in phase (ii), applications are switched to the new C_a, the current configuration at the data mapping component is overwritten with new C_c and the ECUs resume their communication according to the new configuration.

E. State Transition

In order to ensure safe functioning of the applications, especially during reconfiguration, we propose a state management component to monitor and control the states of the whole system. The whole system can be described by one of the following states:

- **Normal Operation (NO):** The whole system runs normally with current configuration deployed both for the applications and the middleware layer. No reconfiguration request has been made.
- **Configuration Calculation (CC):** A reconfiguration request has been triggered by a certain event and sent to the configuration calculator. The calculator is currently synthesizing the new configuration. The applications and the data mapping component run normally with current configuration.
- **Configuration Deployment: (CD):** A valid new configuration has been calculated and passed on to the deployment management components of the relevant ECUs. During configuration deployment, the applications and mapping component function according to the old configuration.
- **Wait for Reconfiguration (WR):** The deployment manager of all relevant ECUs now hold the new valid configuration. All ECUs waits for the time point to perform a synchronized reconfiguration.

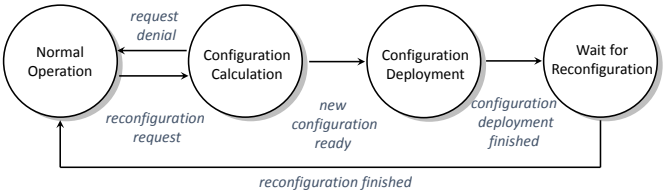

Fig. 5: State transition diagram.

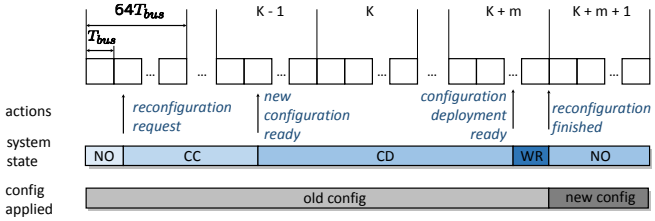

Fig. 6: Timing diagram for state transition.

Fig. 5 demonstrates the state transition diagram where the transitions are labeled by *actions* triggering a change of state. Fig. 6 shows the state transition during a reconfiguration process aligned with the FlexRay communication cycles. As already mentioned, we consider the case where each ECU is synchronized to the FlexRay controller and the middleware can thus obtain the FlexRay cycle counter. Once a reconfiguration request is triggered, the configuration calculator takes the request and starts to calculate a valid new configuration. The process of synthesizing this new configuration may take a considerable amount of time, much larger compared to the FlexRay communication cycles. Let us assume that a new configuration is synthesized and available sometime in the $(K-1)$th 64-cycle sequence. Then the ECU where the calculator is mapped will start transmitting the new configuration as a message to all relevant ECUs from the Kth sequence. We consider the case where m sequences are necessary for the transmission of the full configuration. Once this transmission is finished, the system waits for the $(K+m)$th sequence to finish. At the beginning of the $(K+m+1)$th sequence of 64-cycle, all relevant ECUs will simultaneously use the new configuration and the whole system goes into the (NO) state.

F. Reconfiguration Request

There could be multiple reasons for a reconfiguration request. One example is that a newly installed application or an inactive application needs to be activated online and therefore the system needs to allocate communication resources online. Conversely, a currently active application can be switched off and the resources can be utilized by other applications to achieve better performance. Another example is that a currently active application needs to be switched to a mode offering higher performance. In all such cases, an event would trigger the reconfiguration request manager to send a request to the configuration calculator and the calculator will then start synthesizing the new configuration based on the request and the manifest of the applications. Such a request contains information about which applications should be active, and the requested mode, if an application is requested to switch to a specific mode.

G. Configuration Calculation

The configuration calculator is a software component that computes the configuration C_a and C_c using a mathematical model of the mapping problem. Once the calculator receives the request for a reconfiguration, it will start computing the configuration according to the request and the application manifest. If a new valid configuration is obtained, it will pass this to the deployment management component and deploy the new configuration. If the new configuration is identical to the old one or no valid configuration can be obtained, no reconfiguration process will take place. This component can be implemented as a task which can be mapped on any ECU, provided the ECU can accommodate the transmission of the configuration. But this task usually takes much longer than application tasks and can last for a number of FlexRay communication cycles or application periods. Depending on the operating system used (e.g., an non-preemptive operating system), complications may arise when this task interferes with other application tasks. In that case, the configuration calculator can be mapped on a separate ECU.

Mathematical modeling: Since the mapping of messages to communication resources is independent for each ECU, the whole problem can be divided into subproblems for each single sending ECU. Without loss of generality, we describe only the modeling of the problem for a specific ECU. We consider that a set of applications $a_i \in \mathcal{A}$ are sending messages. Then C_a can be defined as $C_a = \{\alpha_i | a_i \in \mathcal{A}\}$. As discussed in the subsection of data mapping, we put all messages of one application together into a large message, C_c can be defined as $C_c = \{m_i(\alpha_i) | a_i \in \mathcal{A}\}$, where $m_i = (w_i, p_i, (S_i, B_i, R_i))$. Since p_i can be reasonably assumed to be a multiple of the communication cycle, so $p_i = R_i T_{bus}$. Therefore we can simplify the C_c into $C_c = \{w_i, S_i, B_i, R_i\}$. Additionally, w_i and p_i (thus R_i) will be obtained from manifest once α_i is known. The input of the problem include (i) underlying communication resources, i.e., $S_j \in \mathcal{S}$, where $\mathcal{S}$ denotes the set of static slots assigned to the ECU, (ii) the active applications and the requested mode if any, (iii) the application manifest. This problem can be solved using either an *Integer Linear Programming* (ILP) formulation or a simple linear search method.

ILP formulation: The problem of an ILP formulation to pack messages in to FlexRay slots is a well-studied subject. [8] has transformed this problem into a bin packing problem and provided an ILP formulation for it. However, here we need to consider the case of multi-mode problem, where the data size and repetition rate of messages depend on the mode and thus is difficult for linear formulation. Therefore we divide the problem into two layers. In the upper layer, we traverse possible mode combinations, which are determined by the active applications and mode request. Here we consider a set of mode combinations, where the k^{th} mode combination can be represented as $C_{a,k} = \{\alpha_i\}_k \in \mathcal{C}_a$. The corresponding data width $\{w_i\}_k$, repetition rate $\{R_i\}_k$ and performance value $\{J_i\}_k$ can be obtained from the manifest. We consider the total performance of a mode combination to be $J_{total,k} = \sum \lambda_i J_i$, where λ_i is the weight of a_i. In the lower layer, for each mode combination, we will use the bin-packing problem according to [8] to solve for feasible schedules. The lower layer ILP problem can be formulated as follows. The size of a bin is the same as the size of one static slot: $W \times H$, where W is the payload length of a slot and $H = 64$ is the number of communication cycles in a sequence. The transformation from slot-packing to a bin-packing problem [8] converts a message m_i into a rectangular element of size $h_i w_i$. The height of each message is related to the repetition rate as: $h_i = H/R_i$. If y_i is the offset of this message on the y-axis of the bin, the *level* of the message is given by $l_i = y_i/h_i$. In contrast to a common bin packing problem, the above transformation has some additional constraints: height of all elements are a power of two, height of the bin is at least the maximal height of all elements and each element can be placed only on a multiple of its height on the y-axis, i.e., $y_i = l_i h_i$. A set of binary variables $\{\gamma_{i,s,l}\}$ denoting m_i is mapped on slot s and at level l are introduced. The constraints of the problem can be formulated as

$$\forall a_i \in \mathcal{A}, \sum_{s \in \mathcal{S}, l \in \{0,\dots,R_i-1\}} \gamma_{i,s,l} = 1 \tag{1}$$

$$\forall s \in \mathcal{S}, y \in \{0, ..., H-1\} \sum_{a_i \in \mathcal{A}} w_i \gamma_{i,s,\left\lfloor \frac{y}{h_i} \right\rfloor} \leq W \tag{2}$$

where constraint (1) forces each message to occur exactly once among all bins (static slots) and constraint (2) ensures that each bin is not overloaded. Here, instead of minimizing the slots used, we only need a feasible schedule. If a feasible schedule for the k^{th} mode combination exists, the solver will return values of slot S_i and level l_i for each message and B_i can be obtained from l_i. If the k^{th} mode combination does not allow a feasible schedule, the corresponding performance value is set to zero, i.e., $J_{total,k} = 0$. The mode combination C_a with highest performance and its corresponding C_c is returned as the output of this component.

Linear search: Alternatively, the lower layer ILP formulation can be replaced by a linear search method, where for each possible mode combination, the values of S_i and B_i is generated and checked for compliance with the constraints and the combination is considered feasible if a valid $C_c = \{w_i, S_i, B_i, R_i\}$ can be obtained.

It should be mentioned that in both cases the computation time will increase greatly as the size of the problem increases. However, as mentioned above, this packing problem is well-studied and more efficient algorithm to solve this problem and the optimization of the computation time is not the focus of this paper.

a_1				a_2				a_3				a_4			
α_1	w_1	p_1	J_1	α_2	w_2	p_2	J_2	α_4	w_3	p_3	J_3	α_4	w_4	p_4	J_4
1	16	T_{bus}	100	1	8	T_{bus}	100	1	16	T_{bus}	100	1	8	T_{bus}	100
2	16	$2T_{bus}$	50	2	8	$2T_{bus}$	56	2	8	T_{bus}	50	2	8	$2T_{bus}$	56
3	8	$2T_{bus}$	25	3	8	$4T_{bus}$	34	3	4	T_{bus}	25	3	8	$4T_{bus}$	34
Off			0	Off			0	Off			0	Off			0

TABLE III: Communication manifest of the applications in the case study.

IV. Experimental Results

A. Case Study

In this section, we use a case study to demonstrate the communication middleware proposed in this paper. We first explain the system setup, how the performance values can be obtained and then a series of steps of reconfiguration requests to be triggered. In the case study, we consider a synthetic system consisting of four applications a_1 to a_4 mapped on two ECUs, E_1 and E_2. Each application has three different modes. Table III shows the communication manifest of the applications, containing the data size per message in bytes, message period and the performance value for each mode. All messages are sent from E_1 to E_2. We consider equal weights for performance values of all applications. If multiple mode combinations have the best performance, the calculator chooses one of them. FlexRay is configured to have a communication cycle of $T_{bus} = 10ms$. Two static slots with a maximal payload of 16 bytes, represented here as s_1 and s_2, are assigned to E_1 statically.

Performance: The performance value of each mode should be specified by the application designer. Here we illustrate how a set of performance values can be obtained. We consider in this case application a_2 and a_4 as control applications and application a_1 and a_3 as realtime applications. The performance for control applications is measured by the control performance and we assume that the performance of the realtime applications are directly proportional to the average amount of data transmitted per FlexRay communication cycle. Then we normalize the performance according to a scale of 100. For the control applications we consider a feedback control system with one sampling period delay. The discrete control system with one sample delay can be represented as

$$x[k+1] = Ax[k] + Bu[k], \tag{3}$$

where $x[k]$ is the feedback states of the plant and $u[k] = Kx[k-1] + Fr$ is the control input for one sample delay. Here r is the reference value, K and F are respectively the feedback and feed forward gain. $y[k] = Cx[k]$ represents the output. Here we consider the integral square error as the control performance, which can be represented as [9]

$$J = \sum ||y[k] - r||^2. \tag{4}$$

The reciprocal of this performance is taken to represent the performance of the control applications. Here we use a DC motor speed control plant model in the case of step response to reference r. The control gains are designed using the method discussed in [10]. We assume that the sensor task is mapped on one ECU while the controller and actuator tasks are mapped on the other. The sensor data are transmitted on the FlexRay bus. For each sampling period, we find a set of gains that optimizes the performance. The performance of each application with respect to the mode obtained through simulation is shown in Fig. 7. In the experiment we will use these performance values.

Reconfiguration requests: Here we demonstrate the function of the proposed middleware by injecting a series of pre-programmed reconfiguration requests, as shown in Table IV, and show the results of the reconfiguration.

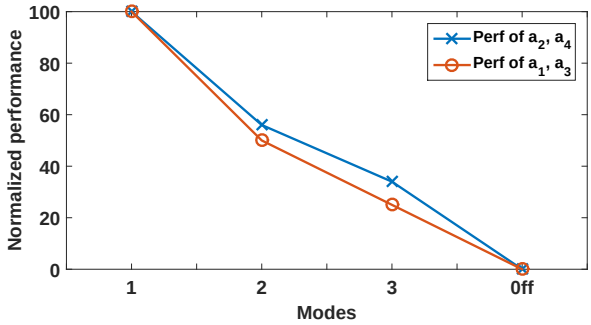

Fig. 7: Performance of the applications for all modes.

B. Hardware Implementation

The case study is implemented in a hardware system using three EB6120 ECUs [11]. Here we use two ECUs to serve as E_1 and E_2. Since we use a non-preemptive operating system, in order to prevent interference on the schedules of application tasks on E_1 and E_2, the configuration calculator is mapped on an extra ECU E_3. The software including a prototype of the proposed communication middleware, is developed using a COTS tool chain - Matlab/Simulink and SIMTOOLS/SIMTARGET toolbox [12]. In this tool chain, software for application tasks are developed using Simulink blocks and the SIMTOOLS/SIMTARGET toolbox is used to provide interfaces between the application software and the underlying operating system and the FlexRay communication. These interfaces include the schedules of tasks, the mapping of signals onto FlexRay frames, etc.. The FlexRay communication is configured using the SIMTOOLS blocks. Then SIMTARGET is used to generate binary files for the ECUs from the Simulink models. The middleware for the case study is implemented with Simulink blocks including Matlab embedded function blocks. The configuration calculator is implemented according to the linear search method and for the framing in the data mapping component, we adopted the alternative without the index field.

C. Results and Discussions

Table IV shows the reconfiguration of the system according to the series of request steps. The performance values of applications for each step are shown in Fig. 8. From this table and figure, we can observe the application mode switch in the whole series of reconfiguration steps. From step 1 to step 3, a_1 to a_3 are incrementally switched on. In step 1 and step 2, the assigned slots can accommodate the applications in their best mode and the overall performance value rises. In step 3, when a_3 is switched on, not all applications can run in their best mode due to inadequate resources. Therefore the configuration calculator has synthesized a configuration placing a_1, a_2 and a_3 respectively in modes 1, 1 and 2.

Steps	Request	a_1			a_2			a_3			a_4			System	
		α	$\{S,B,R\}$	J	α	$\{S,B,R\}$	J	α	$\{S,B,R\}$	J	α	$\{S,B,R\}$	J	J_{total}	$J_{avg.}$
0		off		0	off		0	off		0	off		0	0	0
1	$a_1 \rightarrow$ on	1	$\{s_1,0,1\}$	100	off		0	off		0	off		0	100	100
2	$a_2 \rightarrow$ on	1	$\{s_2,0,1\}$	100	1	$\{s_1,0,1\}$	100	off		0	off		0	200	100
3	$a_3 \rightarrow$ on	1	$\{s_2,0,1\}$	100	1	$\{s_1,0,1\}$	100	2	$\{s_1,0,1\}$	50	off		0	250	83
4	$\alpha_3 \rightarrow 1$	3	$\{s_2,0,2\}$	25	1	$\{s_2,0,1\}$	100	1	$\{s_1,0,1\}$	100	off		0	225	75
5	$a_4 \rightarrow$ on	3	$\{s_1,1,2\}$	25	1	$\{s_1,0,1\}$	100	1	$\{s_2,0,1\}$	100	2	$\{s_1,0,2\}$	56	281	70
6	$\alpha_4 \rightarrow 1$	3	$\{s_1,1,2\}$	25	2	$\{s_1,0,2\}$	56	1	$\{s_2,0,1\}$	100	1	$\{s_1,0,1\}$	100	281	70
7	$a_3 \rightarrow$ off	1	$\{s_2,0,1\}$	100	1	$\{s_1,0,1\}$	100	off		0	1	$\{s_1,0,1\}$	100	300	100
8	$a_3 \rightarrow$ on	3	$\{s_1,1,2\}$	25	1	$\{s_1,0,1\}$	100	1	$\{s_2,0,1\}$	100	2	$\{s_1,0,2\}$	56	281	70

TABLE IV: Synthesized configuration and performance values for the reconfiguration request steps in the case study.

In step 4, a_3 is forced to run in mode 1, and taking this into account, a_1 is switched down to mode 3 to release resources for a_3. Note that in this case, the overall performance value drops from 250 in step 3 to 225. This is because the configuration synthesized here is not the one with the best overall performance, since it is constrained that a_3 has to run in mode 1. Then when additionally a_4 is activated and the condition for a_3 is removed, the whole system switched to a case where the four applications run respectively in mode 3, 1, 1, 2, which offers the best overall performance value when all 4 applications are active. Similar to step 4, in step 6, a_4 is forced to run in mode 1, and to reallocate the communication resources for this, a_2 is switched down to a mode with lower performance. Step 7 demonstrates the behavior of the system when an application is deactivated. When a_3 is switched off, the rest of the applications would be able to switch to their best mode. Note that here the overall performance is better than in step 5 and step 6. This is because such a mode combination is not available for step 5 and 6, since there all 4 applications need to be active. In the final step, when a_3 is switched on again, the whole system returns to the configuration in step 5, where the requirement is identical. Fig. 9 shows the average time taken on hardware to calculate the configuration for each reconfiguration step based on our implementation. Step 5 and 8 takes considerably longer because the the calculator has to go through all combinations of 4 applications. But it can be observed that the configuration can be calculated online within a reasonable amount of time for all steps in the case study.

In this case study, the communication resources available for the four applications are not sufficient for all applications to run in their best mode. From the experimental results, we can see that the proposed communication middleware can be reconfigured to reallocate the communication resources within an ECU to enable resource sharing between multi-mode applications in a FlexRay-based distributed system at runtime. The configuration can be synthesized and deployed according to a request and optimize the overall performance while complying to the constraints specified in the request.

V. Concluding Remarks

This paper demonstrates a middleware layer and its implementation using commercial tools, that allow a dynamic reconfiguration of applications whose messages are scheduled on the static segment of FlexRay. Future work may generalize this idea by extending the middleware layer to accommodate scheduling of application data on the underlying bus following any communication protocol. This will allow selection of the type of communication medium used by an application at runtime, which will increase the fault-tolerance of the medium along with better optimization of the performance of applications. Furthermore, we would explore more computational efficient algorithms for the online configuration calculation.

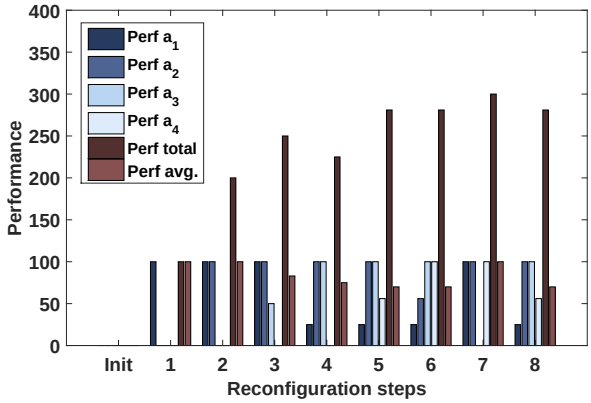

Fig. 8: Performance value of each application and the overall performance in all reconfiguration steps.

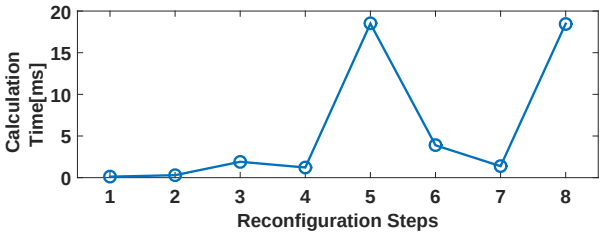

Fig. 9: Average time for configuration calculation on hardware for all reconfiguration steps.

References

[1] P. Mundhenk, F. Sagstetter, S. Steinhorst, M. Lukasiewycz, and S. Chakraborty, "Policy-based message scheduling using flexray," in *CODES+ISSS*, 2014, pp. 1–10.

[2] L. T. Phan, I. Lee, and O. Sokolsky, "A semantic framework for mode change protocols," in *RTAS*, 2011, pp. 91–100.

[3] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham, "Mode change protocols for priority-driven preemptive scheduling," *Real-Time Systems*, vol. 1, no. 3, pp. 243–264, 1989.

[4] J. Real and A. Crespo, "Mode change protocols for real-time systems: A survey and a new proposal," *Real-time systems*, vol. 26, no. 2, pp. 161–197, 2004.

[5] G. Fohler, "Changing operational modes in the context of pre run-time scheduling," *IEICE Transactions on Information and Systems*, vol. 76, no. 11, pp. 1333–1340, 1993.

[6] K. Klobedanz, A. Koenig, and W. Mueller, "A reconfiguration approach for fault-tolerant flexray networks," in *DATE*, 2011, pp. 1–6.

[7] "Flexray communications system protocol specification, version 2.1," *www. flexray. com*, 2005.

[8] M. Lukasiewycz, M. Glaß, J. Teich, and P. Milbredt, "Flexray schedule optimization of the static segment," in *CODES+ISSS*, 2009, pp. 363–372.

[9] R. Schneider, D. Goswami, S. Zafar, M. Lukasiewycz, and S. Chakraborty, "Constraint-driven synthesis and tool-support for flexray-based automotive control systems," in *CODES+ISSS*, 2011, pp. 139–148.

[10] D. Goswami, R. Schneider, and S. Chakraborty, "Relaxing signal delay constraints in distributed embedded controllers," *IEEE Trans. Contr. Sys. Techn.*, vol. 22, no. 6, pp. 2337–2345, 2014.

[11] *Elektrobit. www.elektrobit.com.*

[12] *SIMTOOLS/SIMTARGET. www.simtools.at.*