Refactoring an Automotive Embedded Software Stack using the Component-Based Paradigm

Thomas M. Galla*, Dietmar Schreiner†‡, Wolfgang Forster*,
Christof Kutschera†, Karl M. Göschka‡, and Martin Horauer†
*DECOMSYS
Dependable Computer Systems GmbH
Stumpergasse 48 / 28, A-1060 Vienna, Austria
Email: wolfgang.forster@decomsys.com
Email: thomas.m.galla@decomsys.com
†University of Applied Sciences Technikum Vienna
Department of Embedded Systems
Höchstädtplatz 5, A-1200 Vienna, Austria
Email: kutschera@technikum-wien.at
Email: horauer@technikum-wien.at
‡Vienna University of Technology
Institute of Information Systems, Distributed Systems Group
Argentinierstrasse 8 / 184-1, A-1040 Vienna, Austria
Email: d.schreiner@infosys.tuwien.ac.at
Email: k.goeschka@infosys.tuwien.ac.at

Abstract— **The number of electronic systems in cars is continuously growing. Electronic systems, consisting of so-called electronic control units (ECUs) interconnected by a communication network, account for up to 30% of a modern car's worth. Consequently, software plays an ever more important role, both for the implementation of functions and the infrastructure.**

In order to benefit from the reuse of software modules, the major automotive companies have standardized a large number of these modules in the context of the AUTOSAR consortium.

In this paper we propose the refactoring of the AUTOSAR stack of system software modules by applying the component-based paradigm in order to increase the scalability of the software stack according to the particular requirements of the application. We demonstrate the feasibility of this approach by performing the refactoring of the modules FlexRay Driver and FlexRay Interface as an example and by deploying the resulting refactored components in a sample automotive application. Finally, we measure the execution time as well as the memory consumption of the refactored components and compare these measures to the measures obtained from the corresponding ordinary AUTOSAR modules.

I. Introduction and Related Work

In the last decade the percentage of electronic components in today's cars has been ever increasing. According to [1] the new S-Class Mercedes for example utilizes seven communication buses and 72 microcontrollers.

Since 1993 major automotive companies have been striving for the deployment of standard software modules in their applications since the potential benefits are huge [2]. This trend has been a key motivation for the formation of the AUTOSAR [3] consortium in 2002. Important issues in this context are safety (increased test depth of standard software modules), software reuse, for the possibility to combine software modules supplied by different vendors due to standardized interfaces, and—last but not least—cost reasons in order to cope with shorter development cycles.

The software stack proposed by AUTOSAR follows a layered architecture of basic software modules comprising communication modules, operating system, and modules providing access to the microcontroller's integrated peripheral devices (e.g., A/D converters, digital I/O).

In this paper we will reason that a layered architecture as proposed by AUTOSAR is inefficient as far as resource usage is concerned (especially memory consumption) due to limited adaptivity to the needs of the application software. In order to overcome this drawback, we will propose a refactored version of the AUTOSAR stack of basic software modules by applying the component-based paradigm [4], [5] to the AUTOSAR stack. Hereby we will focus on a well-defined part of the AUTOSAR software stack namely the software modules related to the FlexRay communication system [6], [7]. This approach is fundamentally different to most component-based approaches which, as far as communication is concerned, rely on the existence of a (non component-based) middleware. The approach presented in this paper, on the contrary, addresses communication in a component-based fashion as well, by modeling this communication by means of explicit connectors [8]. This approach gives the possibility to choose the type of connector best suited for the given application and thus reduces the memory footprint as well as the communication overhead compared to a monolithic non component-based middleware. This selection of the best-suited

1-4244-0840-7/07/$20.00 ©2007 IEEE.

connector is based on so-called contracts which govern the selection during a model transformation process [9]. Finally, by means of a sample application, we will show that the refactored version of the AUTOSAR software modules into a set of finer grained software components provides better scalability and thus reduces the overall resource consumption of the software stack for a specific application.

The paper is structured as follows: Section II illustrates the AUTOSAR hard- and software architecture. Section III describes how the component-based approach is applied to the AUTOSAR FlexRay communication modules. In Section IV the refactored component-based AUTOSAR FlexRay communication stack is deployed in a sample application. The resulting memory footprint and the execution time of the refactored stack is analyzed and compared to an ordinary AUTOSAR stack applied to the very same application. Section V gives a short summary of the results and concludes the paper.

II. System Architecture

A. Hardware Architecture

The hardware architecture of automotive systems can be viewed at different levels of abstraction. On the highest level of abstraction, the *system level*, an automotive system consists of a number of networks interconnected via gateways. In general these networks correspond to the different functional domains that can be found in today's cars (i.e., chassis domain, power train domain, body domain).

The networks themselves comprise a number of electronic control units (ECUs) which are interconnected via a communication media. The physical topology used for the interconnection is basically arbitrary; however, bus, star, and ring topologies are the most common topologies in today's cars. – This *network level* represents the medium level of abstraction.

On the lowest level of abstraction, the *ECU level*, the major parts of an ECU are of interest. An ECU comprises one or more micro controller units (MCUs) as well as one or more communication controllers (CCs). In most cases, exactly one MCU and one CC are used to build up an ECU. In order to be able to control physical processes in the car (e.g., control the injection pump of an engine) the ECU's MCU is connected to actuators via the MCU's analogue or digital output ports. To provide means to obtain environmental information, sensors are connected to the MCU's analogue or digital input ports. We call this interface the ECU's environmental interface. The CC(s) facilitate(s) the physical connectivity of the ECU to the respective network(s). We call this interface of an ECU the ECU's network interface.

B. Software Architecture

The AUTOSAR software architecture makes a rather strict distinction between application software and basic or system software. While the *basic (or system) software* provides functionality like communication protocol stacks for automotive communication protocols (e.g., FlexRay [6], [7]), an operating system and diagnostic modules, the *application software* comprises all application specific software items (i.e., control loops, interaction with sensor and actuators). This way, the basic or system software provides the fundament the application software is built upon.

The *Runtime Environment (RTE)* provides the interface between application software components and the basic software modules as well as the infrastructure services that enables communication to occur between application software components.

1) Application Software Architecture: Application software in AUTOSAR consists of application software components, which are ECU and location independent and sensor-actuator components that are dependent on ECU hardware and therefore location dependent. Whereas instances of application software components can easily be deployed to and relocated among different ECUs, instances of sensor-actuator components must be deployed to a specific ECU for performance/efficiency reasons. Deploying multiple instances of the same component to a single ECU is supported by the AUTOSAR component standard.

Application software components as well as sensor-actuator components are interconnected via *connectors*. These connectors represent the exchange of signals or the remote method invocations among the connected components.

2) System Software Architecture: In addition to the application software components, AUTOSAR also defines a layered architecture of system software modules, which provide a basic platform for the execution of the application software components. Figure 1 gives a coarse grained overview of the major categories of system software modules.

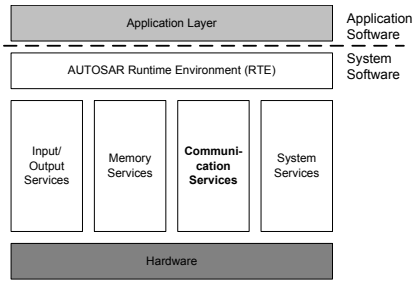


Fig. 1. AUTOSAR – System Software Stack Overview

The *Input/Output Services* are software modules that provide standardized access to sensors, actuators and ECU onboard peripherals (e.g., D/A or A/D converters etc.). The *Memory Services* comprise software modules that facilitate the standardized access to internal and external non-volatile memory for means of persistent storage. The *Communication Services* category, which is of primary interest for the remainder of this paper, contains software modules that provide standardized access to vehicle networks (i.e., the Local Interconnect Network (LIN) [10], the Controller Area Network (CAN) [11], [12], and FlexRay). Last but not least, the *System Services* encompass all software modules that provide standardized (e.g., operating system, timer support, error loggers) and ECU specific (ECU state management, watchdog management) system services and library functions.

The structure of the Communication Services for the FlexRay communication system are depicted in Figure 2.

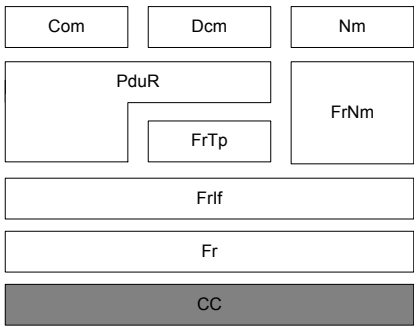


Fig. 2. AUTOSAR – FlexRay Communication Services

The *FlexRay Transport Protocol* module (FrTp) is used to perform segmentation and reassembly of large protocol data units (PDUs)—also termed "messages"—transmitted and received by the Diagnostic Communication Manager (see below). This protocol is rather similar or even compatible (in certain configuration settings) to the ISO TP for CAN [13] specified in ISO/DIS 15765-2.2.

The *PDU Router* module (PduR) provides two major services. On the one hand it dispatches PDUs received via the underlying interfaces (e.g., FlexRay Interface) to the different higher layers (COM, Diagnostic Communication Manager). On the other hand the PDU router performs gateway functionalities between different communication networks by forwarding PDUs from one interface to another of either the same (e.g., FlexRay to FlexRay) or of different type (e.g., CAN to FlexRay).

The *COM* module provides signal-based communication to the higher layers (RTE). The signal-based communication service of COM can be used for intra-ECU communication as well as for inter-ECU communication. In the former case, COM mainly uses shared memory for this intra-ECU communication, whereas for the latter case at the sender side COM packs multiple signals into a PDU and forwards this PDU to the PDU router in order to issue the PDU's transmission via the respective interface. —On the receiver side, COM obtains a PDU from the PDU router, extracts the signals contained in the PDU and forwards the extracted signals to the higher software layers.

The *Diagnostic Communication Manager* module (Dcm) provides services which allow a tester device to control diagnostic functions in an ECU via the communication network (i.e., CAN, LIN, FlexRay). Hereby the Dcm supports the Keyword Protocol 2000 (KWP2000) [14] standardized in ISO/DIS 14230-3 and the Unified Diagnostic Services (UDS) protocol [15] standardized in ISO/DIS 14229-1.

Network management modules provide means for the coordinated transition of the ECUs in a network into and out of a low-power (or even power down) sleep mode. AUTOSAR NM is hereby divided into two modules: a communication protocol independent module named *Generic NM* (Nm) and a communication protocol dependent module named *FlexRay NM* (FrNm).

Based on the frame-based services provided by the FlexRay Driver (see below) the *FlexRay Interface* module (FrIf) facilitates the sending and the reception of protocol data units (PDUs). Hereby multiple PDUs can be packed into a single frame at the sending ECU and have to be extracted again at the receiving ECU. The point in time when this packing and extracting of PDUs takes place is governed by the temporal scheduling of so-called communication jobs of the FlexRay Interface. The instant when the frames containing the packed PDUs are handed over to the FlexRay Driver for transmission or retrieved from the FlexRay Driver upon reception is triggered by communication jobs of the FlexRay Interface as well. Hereby each communication job can consist of one or more communication operations, each of these communication operations handling exactly one communication frame (including the PDUs contained in this frame).

Just like the FlexRay Interface module, the *FlexRay Driver* module (Fr) is protocol specific as well. The FlexRay Driver module provides the basis for the FlexRay Interface module, by facilitating the transmission and the reception of frames via the respective communication controller.

III. Applying the Component-Based Approach

Components may interact at runtime if their related provided and required interfaces are validly associated at composition time. This association, namely the connector, is an abstract representation of any interaction occurring between the connected components. In most component models the process of component interaction is covered within some middleware, therefore we consider those kinds of connectors to be implicit.

An explicit connector on the other hand is an architectural entity that is used to represent component composition and interaction and owns its unique implementation of interaction operators. Therefore, an explicit connector encapsulates all communication logic for one specific type of interaction. In addition an explicit connector specifies properties of the connected components' interaction and provides contracts [16], [17] regarding communication channels and resource requirements. These contracts define guarantees about the behavior of the associated elements, by specifying requirements and provisions of associated elements. In general a contract consists of two obligations:

1) A client, requiring a service from a provider, has to satisfy the provider's preconditions.
2) A provider of that required service has to fulfill its postcondition, if the client's precondition is met.

Hereby we distinguish five types of contracts [8] that are named after the model element they are associated with:

1) *Component-contracts* deal with a component's resource requirements or deployment restrictions like required memory or required ECU type.
2) *Interface-contracts* specify services and properties of the components' interfaces like operation signatures,

interface type or temporal properties like worst-case execution time (WCET) at operation level.

3) *Port-contracts* deal with the relation between component ports and interfaces. Behavioral protocols are typically contained within port-contracts.
4) *Connector-contracts* specify constraints related to the used communication channels like worst-case propagation delays, but also regarding resource requirements of the connector implementation.
5) *Platform-contracts* specify properties of platform elements like ECUs or communication systems e.g. ECU type, available memory or timing information.

These contracts, when associated with explicit connectors introduced previously, govern the model transformation process, where explicit connectors are transformed into components representing so-called connector fragments. —This transformation process is described in Section III-C in more detail.

Applying the component-based paradigm to the AUTOSAR system software, however, is not feasible without performing an adequate refactoring of the AUTOSAR system software modules in advance. This refactoring yields finer-grained system software components exhibiting a well-defined functionality and thus provides the basis for an application dependent selection of only those components needed by a specific application. This refactoring consist of two steps—namely *vertical layer slicing* and *combining of the module slices of adjacent layers*—which are discussed in the following subsections.

Note that this approach is fully compliant with AUTOSAR, since AUTOSAR defines different conformance classes, and allows (in certain conformance classes) the grouping of several modules into module groups, where the interfaces of the module group with other AUTOSAR modules (which are not a member of this group) have to adhere to the AUTOSAR specification, whereas those interfaces which are only module group internal (intra module group interfaces) do *not* have to adhere to the AUTOSAR specification.

A. Vertical Layer Slicing

The first step of transforming the AUTOSAR layered module software architecture into a component based software architecture is the vertical slicing of the different modules into module slices. We hereby assume that each module comprises a number of rather independent functional units. The process of slicing must be governed by the following premises:

- Related functional units shall be located in the same module slice.
- The number of interactions between the functional units of different module slices must be minimized.
- The functional units within a single module slice shall exhibit a large amount of inter functional unit interaction.

This way, we arrive at a defined set of module slices, each slice consisting of one or more functional units, where the module slices exhibit a low number of inter-unit interaction but a high number of intra-unit interactions.

The key idea behind this slicing is that each of the resulting module slices is a first class candidate for becoming (a part of) a dedicated system software component.

1) Slicing of the AUTOSAR FlexRay Interface and Driver Module: When applying the slicing step to the AUTOSAR FlexRay Driver, we identified the following module slices for the AUTOSAR FlexRay Driver: A *base slice* containing all the functional units that provide some kind of basic functionality to the module (e.g., module initialization). — This slice is always required (and thus must be deployed) when using any of the other slices of the module. All functional units dealing with FlexRay's global time (e.g., alarm timer services based on this global time) have been allocated into a *time services slice*. Functional units dealing with FlexRay's wakeup service have been grouped into a *wakeup slice*, all functional units related to the handling of FlexRay's media test symbols have been combined in a *media test symbol slice*, and functional units for querying the operational status of the FlexRay communication controller (e.g., checking whether the communication controller is synchronous with the other controllers in the FlexRay network) have been combined in a *status slice*. The functional units dealing with the transmission of PDUs (in the FlexRay Interface) or frames (in the FlexRay Driver) have been combined in a *transmission slice* whereas the functional units dealing with reception have been allocated in a *reception slice*.

2) Slicing of the AUTOSAR FlexRay Transport Protocol: Aside from the obligatory base slice which provides basic functionality of the module the AUTOSAR FlexRay Transport protocol mainly consists of five almost orthogonal functional units, namely a unit dealing with *segmentation and reassembly* if the data to be transmitted exceeds the maximum transfer unit of the underlying communication protocol, a functional unit dealing with *acknowledgment*, a functional unit for *flow control handling*, and last but not least, functional units for *reception* and *transmission*.

Since these functional units implement to a large degree orthogonal functionality, almost all 2^5 possible combinations of the functional units are feasible.

One possible combination (which is a typical use case in diagnostic communication using ISO/DIS 14229-1) is non-segmented, non-acknowledged[1], communication without flow control. – In this combination it is obvious that the functional units dealing with flow control, segmentation, and acknowledgment are not required and can thus be omitted from the system software stack. Furthermore, if an ECU is only a receiver in diagnostic communication, the sending functional unit can be eliminated as well.

B. Combining Module Slices of Different Layers

After the slicing of the modules has been performed, further improvements with respect to execution time and memory

[1]No acknowledgment on transport layer level is used, since acknowledgment takes place on the diagnostic layer level (which is one layer above the transport layer) anyway.

consumption can be achieved, when combining related slices of vertically adjacent modules into a single component.

Again this combination shall be guided by the premises already stated in Section III-A.

Reasonable candidates for such a combination are the reception and the transmission slices of the FlexRay Driver with those of the FlexRay Interface, since every time, a higher layer of the system software stack used the transmit functionality of the FlexRay Interface, the transmit functionality of the FlexRay Driver is required as well.

When looking at the AUTOSAR FlexRay Transport Protocol however, the situation gets a little bit more complex. —For the transmission of transport layer messages, the transmit slices of FlexRay Driver and FlexRay Interface are required. Furthermore, if acknowledgment or flow control is required by the higher layer, the respective slices (namely the flow control and/or the acknowledgment slice) of the FlexRay Transport Protocol are required. Since both functionalities require communication from the receiving to the sending ECU, the reception slices of FlexRay Interface and FlexRay Driver are needed as well.

C. Model Transformation

As already stated, we build component based applications using a model driven development approach. Interaction between application components is expressed by explicit connectors. Therefore, the first required step in developing a component based application is to define a platform independent model (PIM) of the desired component architecture: All used components and all their interdependencies modeled, by means of explicit connectors, are specified within a composition specification. At this phase of development, the explicit connector is an abstract representation of communication and interaction requirements that provides little information on the runtime properties of the process of interaction. In the second step, a platform specific model (PSM) of the deployment specification for the application has to be defined. Now that information on the target platform and the component's physical location is specified, the nature of available communication channels becomes visible. At this step one can see, for example, if interaction is distributed or local.

By applying model transformations, the abstract explicit connectors from the platform independent composition specification get replaced by a composite structure of component-like model entities. This is done by splitting the abstract connector up into two associated pieces, the connector fragments that are composed elements themselves. Each fragment is attached to exactly one of the components, the original explicit connector was bound to and has to be deployed along with it. The fragment's internal structure is determined by all contracts applied to the explicit connector and to system specifications within the application models. It is obvious that explicit connectors for local compositions may be as simple as local procedure calls and therefore would not require an explicit representation. On the contrary, connectors between components deployed in a distributed manner are represented by rather complex composite structures dealing with matters of concurrency and distributed communication. The whole transformation is recursively applied to the composition model until the resulting model contains no other connectors than local procedural ones (that represent local procedure calls) connecting components or connector fragments.

Beside the additional contracts, that arise by introducing new model elements, the connector fragments respectively their internals, that can be used to improve the verification of the application model, our approach also issues great potential for optimization of the application's communication subsystem. As mentioned before, a connector fragment is a composite structure itself and contains only that communication primitives it requires to fulfill operations specified within the application model. Therefore, all features provided by a generic communication stack that are not required for a specific application can be eliminated from the system.

As an example (see Figure 3) consider an explicit data broadcast connector (C) for distributed deployment that connects two application software components (A and B), where the interface of the sending application software component has a contract assigned that the amount of data sent by this component is smaller than the maximum transfer unit (MTU) of the underlying communication protocol (FlexRay). Consider further that the direction of data exchange is strictly unidirectional and that the contracts of the sending and the receiving application software component's interface state that neither acknowledgment nor flow control is required. Therefore, the connector fragment at the sending application's component side has to contain a request handler and a sender component (for sending access to the FlexRay communication controller) only, whereas the receiving side's fragment has to contain a receiver and a data buffer component only. The selection of the sender and receiver component can be done according to the system specification (e.g., communication media type) and to the interface contract. Therefore, only the components containing the transmission slice of the AUTOSAR FlexRay Transport Protocol are needed. —All other slices of the transport protocol can be eliminated. The remaining slice (the transmission slice) of the AUTOSAR FlexRay Transport Protocol depends on the transmission component of FlexRay Interface and FlexRay Driver.

There are further possible use cases where a large number of module slices can be eliminated during model transformation:

- Most diagnostic services do not require the exchange of data which exceeds the maximum transfer unit of the underlying communication protocol. —Therefore, the segmentation and reassembly module slice of the transport protocol can be eliminated.
- When looking at a simple sensor node which only broadcasts sensor value in signals, one realizes that such a sensor node does not require any transport protocol at all. —From the FlexRay Driver and FlexRay Interface functionality, only the transmission component and the base component is needed.

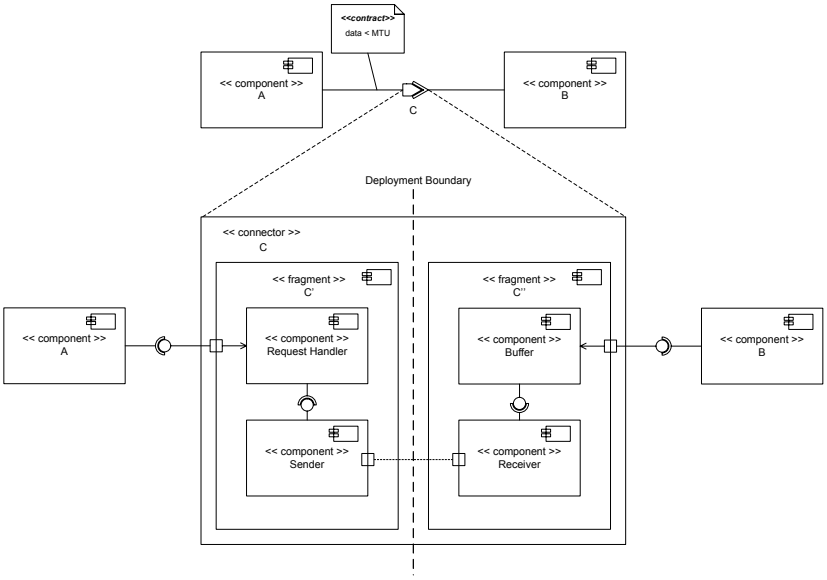


Fig. 3. Example Transformation

IV. Proof of Concept

A. Sample Application

In order to demonstrate the feasibility of the presented approach, we implemented a sample application. This application is based on a typical automotive use case: A central locking system including a speed sensor to provide automatic locking of the doors when the car has reached a certain speed.

The application was specified to contain a central logic for controlling the status of the doors. Inputs are lock and unlock requests (initiated by the user) and the actual speed of the car. Normally the doors are locked and unlocked according to the requests by the user. If, however, the speed of the car exceeds a certain value, the doors are locked automatically by the control logic. As long as the car's speed exceeds the certain value, unlock requests of the user are ignored. The doors are not unlocked automatically when the car slows down below the mentioned speed value; another unlock request has to be received. Every time the doors have to be locked or unlocked, the control logic transmits an according order to each door lock.

1) Composition: The distributed system for the door lock application consists of a sensor which transmits unlock and lock requests as well as a sensor providing the actual speed status. Further, a control logic for processing the sensor data and triggering unlock or lock commands is present. Finally, the distributed system contains actuators locking and unlocking the doors according to the commands of the central control logic.

The key sensors may be implemented at the door lock to detect the turning of a key, at infrared or radio receivers to recognize the usage of a remote control, at buttons in the interior of the car, and so on. I.e., it must be possible to connect more than one key sensor to the system. The requests generated by the key sensors are intended to be used by the lock control logic only.

The speed sensor is considered to provide speed data to every subsystem in need, i.e., one sensor broadcasting the actual speed throughout the car. The actuators for opening and closing the doors may—obviously—have a couple of instances. Therefore the system has to be designed in a way which allows multiple door lock actuators.

The functionality of the application was divided into the following four components: Two sensor components ("key sensor" and "speed sensor"), one controlling component ("lock control") and one actuator component ("door lock").

Multiple instances of the "key sensor" and "door lock" components may be available, i.e., more than one "key sensor" may use the interface to "lock control" as well as "lock control" may use more then one interface instance to door lock components. While designing the example application we focused on one "key sensor" and one "door lock" component instance for reasons of simplicity.

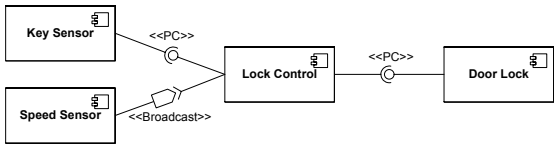


Fig. 4. Composition of the Door Lock Application

As one can see in Figure 4, two *procedure call (≪PC≫)* and one *broadcast (≪Broadcast≫)* interfaces have been chosen. Procedural interfaces may be used if the receiver of the information is explicitly addressed by the sender and the communications purpose is a procedure call (as it is the case in

this application for "key sensor" → "lock control" and "lock control" → "door lock", respectively).

On the other hand, when a component is broadcasting information the receiver(s) is (are) not known—not even if there is a receiver at all. The interface between "speed sensor" and "lock control" is identified as broadcast interface, since the speed sensor just provides the actual speed data, without caring about the identity or the number of receivers.

Figure 4 illustrates this functional decomposition of the door lock application into different components.

2) Deployment: After the application was divided into components, these were assigned to the ECUs where they should be executed. Although "key sensor" and "door lock" components were considered to have multiple instances in a real car, only one instance of each component was arranged for our evaluation. The actual deployment of the components to two ECUs is shown in Figure 5.

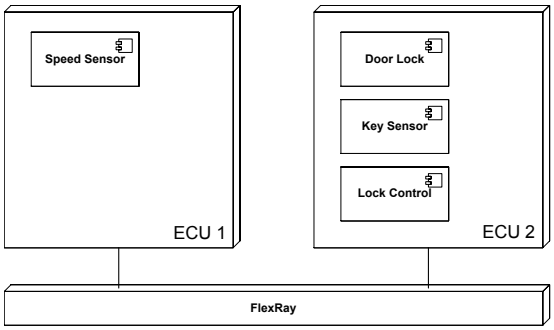


Fig. 5. Deployment for the Door Lock Application

B. Comparison

In order to assess the benefits of the proposed refactoring, we performed the following two comparisons: On the one hand we evaluated the memory consumption of the ordinary FlexRay Interface and the ordinary FlexRay Driver to the respective sliced versions. —Hereby, in the sliced version, only the slices required for the sample door lock application presented in Section IV-A have been considered for the comparison. On the other hand we investigated the memory consumption and the execution time of the combined FlexRay Interface and FlexRay Driver slices in comparison to the respective slices of the ordinary FlexRay Interface and the ordinary FlexRay Driver.

As far as the memory consumption is concerned, both version (original and refactored) have been compiled with the same compiler using the same compiler settings. The memory (RAM and ROM) used has been derived from the compilation output (i.e., from the map file produced in the compilation process).

For the runtime comparison the FlexRay controller's timer (which ticks with a granularity of 1 μs) has been used to time stamp the invocation and the termination of the respective API functions of the transmission and the reception slices.

1) Slicing Module FrIf and Fr: The first evaluation has been targeted at showing the benefit of slicing the ordinary AUTOSAR modules into distinct module slices. —For this purpose, the slicing of the FlexRay Driver and the FlexRay Interface modules has been performed as described in Section III-A.1.

This first evaluation has been conducted on an ARM922T CPU running at 166 MHz and providing 16 bit access to an external ERAY 1.0 FlexRay communication controller.

Figure 6(a) shows the memory consumption (sum of RAM and ROM consumption) of the used module slices of the FlexRay Driver for ECU 1 and ECU 2 of the door lock application compared to the memory consumption of the ordinary FlexRay Driver module for both ECUs. Figure 6(b) shows the same data for the FlexRay Interface. Hereby, the following module slices of the FlexRay Driver and the FlexRay Interface have been deployed to the respective ECUs: The base slice and the transmission slice have been deployed to ECU 1, whereas the base slice and the reception slice have been deployed to ECU 2. The other slices of the FlexRay Interface and the FlexRay Driver are not required for this particular application and have thus not been deployed.

The benefit of this slicing is obvious when looking at Figure 6: For the FlexRay Driver (Figure 6(a)) as well as for the FlexRay Interface (Figure 6(b)), the memory consumption of the sliced version is significantly lower than the memory consumption of the ordinary modules (approximately by 30%).

2) Combining Module Slices of FrIf and Fr: The second evaluation has been targeted at showing the benefit of integrating module slices of different horizontal layers. —For this purpose an integration of the FlexRay Interface transmission slice with the FlexRay Driver transmission slice as well as a merge of the respective reception slices has been performed.

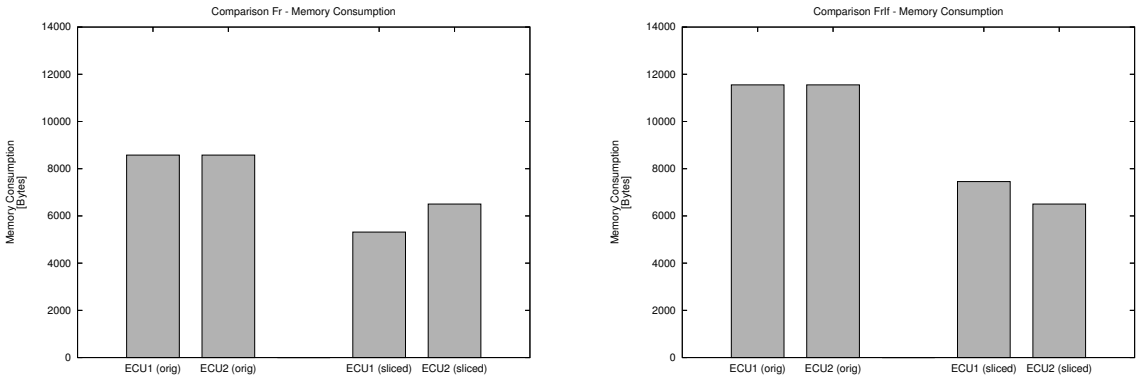
For this second evaluation an NXP SJA2510 N1B microcontroller from NXP Semiconductors[2] with integrated FlexRay communication controller NXPFRDLC running at 80 MHz and providing 32 bit access to the FlexRay communication controller has been chosen. —Program execution took place from the microcontroller's internal flash.

Table I illustrates the key figures, namely the number of PDUs contained in the frame and the total frame length in units of bytes, and the typical use cases in FlexRay networks for the frame layout used in this evaluation.

Figure 7 illustrates a comparison of the sum of the execution times for the plain FlexRay Driver (Fr) and the plain FlexRay Interface (FrIf) to the execution times for the combined slices of FlexRay Driver and FlexRay Interface.

Hereby, Figure 7(a) depicts the minimum executions times whereas Figure 7(b) shows the maximum executions times of the transmission/reception slices of the ordinary FlexRay Interface and the ordinary FlexRay Driver (TX/RX (orig)) as well as of the combined transmission/reception slices (RX/RX (combined)). —It can be seen that the combination has no effect (neither positive nor negative) on the minimum exe-

[2]formerly a division of Philips



(a) FlexRay Driver

(b) FlexRay Interface

Fig. 6. Memory Consumption Comparison

TABLE I
FRAME LAYOUT OVERVIEW

Type	# PDUs	Length	Use Case
1	1	8	Transmission of a CAN-like frame (consisting of 8 bytes)
2	1	16	Transmission of frames with signals for a distributed control loop
3	1	32	Transmission of frames with signals for a distributed control loop
4	3	8	Transmission of multiple CAN-like frames packed into a single FlexRay frame[a]
5	1	254	Transmission of diagnostic frames for in-system flash programming of ECUs

[a]This is a use case for a FlexRay backbone tunneling several CAN frames.

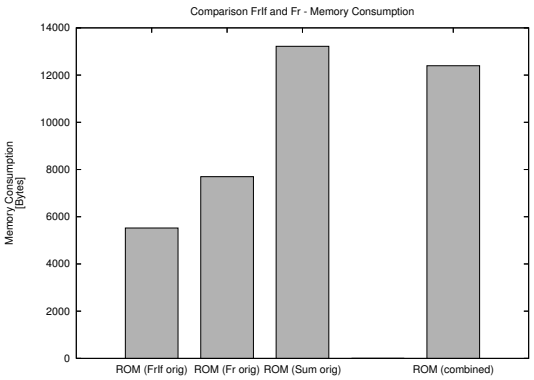


Fig. 8. Comparison Memory Consumption Fr & FrIf

cution time. In the charts depicting the maximum execution times, however, the benefit of combining the respective slices becomes obvious (approximately 10%, see Figure 7(b)).

The effects of this combination with respect to memory consumption can be seen in Figure 8. Looking at the third and fourth bar we see that the memory consumption of the combined FlexRay Interface and FlexRay Driver lies slightly below the sum of the stand-alone ordinary modules.

Note that this decrease in memory consumption gained by combining both modules is independent from the decrease gained by the slicing of the two modules into module slices. — Thus combining both approaches results in increased benefit.

V. CONCLUSION

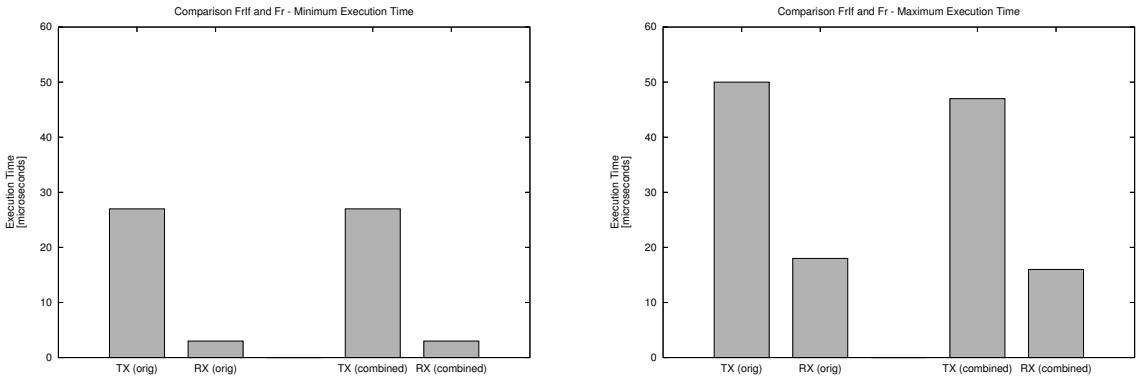
In this paper we presented a refactoring for the AUTOSAR system software stack by applying the component-based paradigm. We demonstrated the feasibility of this approach by performing the refactoring of two sample AUTOSAR modules, namely FlexRay Driver and FlexRay Interface, by implementing the refactored versions, and by deploying these versions in a simple automotive door lock application. We pointed out the benefits of these refactored versions by comparing the memory consumption and the required execution time of the refactored versions to the ordinary AUTOSAR versions. This comparison yielded an approximately 10% reduction in execution time as well as an average 30% reduction as far as the memory footprint is concerned.

The reduction in execution time on the one hand is due to the combination of parts of the FlexRay Driver and the FlexRay Interface module, resulting in a much tighter integration and thus in an elimination of the function call overhead at the interface between these originally separated modules. The decrease in the memory consumption on the other hand is caused by vertical sub-structuring of the original modules into finer-grained module slices and by careful selection of only those slices required for a particular application during deployment.

In the near future, additional benchmarking and an in-depth analysis of the benefits gained by the approaches presented in this paper will be conducted by applying the benchmarking process defined in [18].

ACKNOWLEDGMENT

This work has been partially funded by the FIT-IT [embedded systems initiative of the Austrian Federal Ministry



(a) Minimum

(b) Maximum

Fig. 7. Comparison Execution Time Fr & FrIf

of Transport, Innovation, and Technology] and managed by Eutema and the Austrian Research Agency FFG within project COMPASS [19] under contract 809444.

We would especially like to thank Markus Eggenbauer for implementing the combined FlexRay Driver and FlexRay Interface slices and for conducting the memory and execution time measurements for the comparison.

REFERENCES

[1] P. Hansen, "New S-Class Mercedes: Pioneering Electronics," *The Hansen Report on Automotive Electronics*, vol. 18, no. 8, pp. 1–2, Oct. 2005.

[2] ——, "AUTOSAR Standard Software Architecture Partnership Takes Shape," *The Hansen Report on Automotive Electronics*, vol. 17, no. 8, pp. 1–3, Oct. 2004.

[3] H. Heinecke, "AUTomotive Open System ARchitecture An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures," in *Proceedings of the Convergence International Congress & Exposition On Transportation Electronics*, Detroit, MI, USA, 2004.

[4] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[5] G. T. Heineman and W. T. Councill, Eds., *Component-Based Software Engineering*. Addison Wesley, 2001.

[6] R. Mores, G. Hay, R. Belschner, J. Berwanger, C. Ebner, S. Fluhrer, E. Fuchs, B. Hedenetz, W. Kuffner, A. Krüger, P. Lohrmann, D. Millinger, M. Peller, J. Ruh, A. Schedl, and M. Sprachmann, "FlexRay – The Communication System for Advanced Automotive Control Systems," in *Proceedings of the SAE 2001 World Congress*. Detroit, MI, USA: Society of Automotive Engineers, Mar. 2001.

[7] T. Führer, F. Hartwich, R. Hugel, and H. Weiler, "FlexRay – The Communication System for Future Control Systems in Vehicles," in *Proceedings of the SAE 2003 World Congress & Exhibition*. Detroit, MI, USA: Society of Automotive Engineers, Mar. 2003.

[8] D. Schreiner and K. M. Göschka, "Explicit Connectors in Component Based Software Engineering for Distributed Embedded Systems," in *SOFSEM 2007: Theory and Practice of Computer Science, Proceedings*, ser. LNCS, vol. 4362, Lecture Notes in Computer Science. Springer, Jan 2007, pp. 923–934.

[9] ——, "Synthesizing Communication Middleware from Explicit Connectors in Component Based Distributed Architectures," in *Proceedings of the 6th International Symposium on Software Composition (SC 2007)*, ser. Lecture Notes in Computer Science. Springer, 2007, to appear.

[10] "LIN Specification Package," LIN Consortium, Tech. Rep. Version 2.1, Nov. 2006. [Online]. Available: http://www.lin-subbus.de/

[11] ISO, "Road Vehicles – Controller Area Network (CAN) – Part 1: Data Link Layer and Physical Signalling," ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, Tech. Rep. ISO/DIS 11898-1, 2003.

[12] ——, "Road Vehicles – Controller Area Network (CAN) – Part 2: High-Speed Medium Access Unit," ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, Tech. Rep. ISO/DIS 11898-2, 2003.

[13] ——, "Road Vehicles – Diagnostics on Controller Area Networks (CAN) – Part 2: Network Layer Services," ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, Tech. Rep. ISO/DIS 15765-2.2, April 2003.

[14] ——, "Road Vehicles – Diagnostic Systems – Keyword Protocol 2000 – Part 3: Application Layer," ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, Tech. Rep. ISO/DIS 14230-3, 1999.

[15] ——, "Road Vehicles – Unified Diagnostic Services (UDS) – Part 1: Specification and Requirements," ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, Tech. Rep. ISO/DIS 14229-1, 2004.

[16] B. Meyer, "Applying "Design by Contract"," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.

[17] R. H. Reussner and H. W. Schmidt, "Using Parameterised Contracts to Predict Properties of Component Based Software Architectures," in *Proceedings of the Workshop on Component-based Software Engineering*, S. L. I. Crnkovic and J. Stafford, Eds., 2002.

[18] W. Forster, C. Kutschera, D. Schreiner, and K. M. Göschka, "A Unified Benchmarking Process for Components in Automotive Embedded Systems Software," in *Proceedings of the 10th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2007)*. IEEE, to appear.

[19] "COMPASS: Component Based Automotive System Software." [Online]. Available: http://www.infosys.tuwien.ac.at/compass/