Automotive Software Development for a Multi-Core System-on-a-Chip

Hermann Kopetz, Roman Obermaisser, Christian El Salloum, Bernhard Huber
Vienna University of Technology, Austria
hk@vmars.tuwien.ac.at

Abstract

There are many economic and technical arguments for the reduction of the number of Electronic Control Units (ECUs) aboard a car. One of the key obstacles to achieve this goal is the limited composability, fault isolation and error containment of today's single-processor architectures. However, significant changes in the chip architecture are taking place in order to manage the synchronization, energy dissipation, and fault-handling requirements of emerging billion transistor SoCs (systems-on-a-chip). The single processor architecture is replaced by multi-core SoCs that communicate via networks-on-chip (NoC). These emerging multi-core SoCs provide an ideal execution environment for the integration of multiple automotive ECUs into a single SoC. This paper presents a model-based software development method for designing applications using these multi-core SoCs.

1. Introduction

It is the essence of system and software design to bridge the gap between the requirements of an application on one side and the capabilities of the available hardware base on the other side. Based on the automotive requirements and at-the-time-available hardware devices the following structure of electronic systems aboard a car has evolved over the past twenty years: A highly distributed system consisting of a plurality of heterogeneous Electronic Control Units (ECUs) and intelligent sensors and actuators that are interlinked by a variety of communication networks (CAN, MOST, LIN, etc.). The shape of this structure has been determined, to a significant extent, by concerns about the composability and organizational responsibility for the correct operation of each distributed application subsystem (DAS), e.g., the powertrain system in a car which consists of ECUs for functions like gear control, engine control, and adaptive cruise control.

In order to eliminate any *error propagation path* from one DAS to another DAS, to achieve composability and to reduce the overall system complexity, each DAS is often implemented on its own dedicated distributed hardware base. An ECU is assigned to each job of a DAS and a shared physical communication channel (in the automotive domain often a *CAN* network) is provided for the exchange of the messages within a DAS. We call such an architecture, where each DAS has its own dedicated distributed hardware base, a *federated architecture.*

In the automotive domain the massive deployment of federated architectures has led to the large number of ECUs, sensors, and networks aboard a car. In a typical premium car more than fifty ECUs and five different networks can be found [1]. This large number of ECUs and networks has some undesirable consequences: the high number of cabling contact points (which are a major cause of failures), and the high costs. Furthermore many of these DASes, developed by different organizations, have to communicate in order to share sensor data and provide overall system services.

These challenges can be addressed if a single ECU can host jobs of different DASes. This would significantly reduce the number of ECUs, sensors, networks and cables, improve the sharing of sensor data among the DASes, and lower the installation and maintenance costs. We call such an architecture, where a single integrated hardware base for the execution of jobs from different DASes is provided, an *integrated architecture*. *Hammett* describes aptly the technical challenge in the design of an integrated architecture: [2], p.32: *The ideal future avionics systems would combine the complexity management advantages of the federated approach, but would also realize the functional integration and hardware efficiency benefits of an integrated system.*

In the recent past, a number of efforts have been made to develop such an integrated architecture e.g., *Integrated Modular Avionics (IMA)* [3] in the aerospace domain and *AUTOSAR* [4] in the automotive domain. The key idea in these approaches is the provision of a partitioned operating system for a computer with a single powerful CPU. This operating system is intended to provide in each partition an encapsulated execution environment for a single job of a DAS and eliminate any run-time dependency and error propagation path among the jobs of different DASes. However, the required encapsulation of each job, particularly w.r.t. to temporal properties and transient failures, is difficult to achieve in such an architecture.

Ideally, we would like to see an *integrated system architecture* that has better composability, fault-isolation and error-containment properties than today's *federated* solution, but still supports the integration of multiple functions, developed by different organizational entities, into a single ECU. The advent of properly designed multi-core Systems-on-a-chip (SoCs) offers new possibilities to reach this ambitious goal. This paper explores this design alternative. It focuses on the automotive software development process for a multi-core SoC in order to reduce the number of ECUs and thus realize significant cost-savings and reliability improvements in the electronic systems aboard a car.

This paper is structured as follows. In Section two we describe the proposed model-based software development process and make a case for mapping the ECUs of a federated architecture to the cores of the multi-core SoC that is at the center of the integrated architecture. The details of this proposed SoC are presented in Section three. The integration of software modules of multiple DASes into a single multi-core SoC, i.e., a single novel ECU, is the topic of Section four. Section five explores the composability, fault isolation and error containment properties of this architecture as they relate to the software design process, including some experimental results on the fault-isolation and error containment capabilities. The paper terminates with a conclusion in Section six.

2. The Design Process

We follow a *model-based* design process that extends the one explored in the DECOS project [5]. In model-based design an executable high-level model of an application is at the focus of all further design activities. The behavior of this high level executable model – we call it the *platform independent model (PIM)* – captures and documents the intended application properties (function, timing) of the evolving design without regard to the idiosyncrasies of the targeted execution platform.

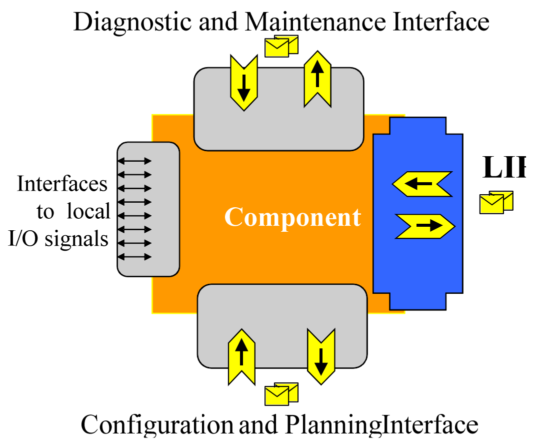
Most large applications do not have a single top, from where the design can proceed in an orderly top-down fashion. They rather consist of a number of nearly independent distributed application systems (DASes) that communicate via gateways. In the automotive domain, the *power train control system*, the *passive safety system*, the *comfort electronics control system* and the *multimedia system* are examples for DASes. We thus propose as a first development step the design of the PIM of each DAS in isolation. In a second development step the different DASes have to be integrated on the selected target platform. For this purpose, a platform-specific (PSM) model of each DAS job that can be executed on the selected target platform has to be generated.

2.1. Distributed Application Subsystems

At the PIM level, a DAS can be represented by a set of *computational components* that exchange messages across dedicated communication channels. A computational component is a time aware entity that consists of a behavioral model – we call it a *PIM-job* – and corresponding execution semantics – we call it the *PIM Container*. The *PIM Container* defines a computational model, e.g., using an abstract machine that executes the high-level programming statements that are used to express the intended behavior at the PIM level. For example, a PIM-job can be a Matlab-Simulink Model that needs the Matlab-Simulink execution semantics (e.g., Stateflow) in order to define the intended behavior. Other PIM-jobs of a DAS may be expressed in JAVA or in the C Programming Language and require the semantics of the JAVA language or the C language. We do not require homogeneous representations of all PIM-jobs of a DAS, since at the architectural level a component is characterized by its interface that hides the implementation.

The proposed design process assumes that the decomposition of a monolithic application into DASes and jobs that can be executed on distributed containers is performed at the PIM level. In many embedded applications, particularly in the automotive domain, this decomposition is a natural part of the problem statement, otherwise a highly distributed automotive architecture, as described in the Introduction of this paper, would not have evolved over the past twenty years. In our view, the critical issue is the *composition* of the jobs of DASes to form an integrated architecture, considering the mandatory constraints of *composability* and *error containment* of each job.

As depicted in Fig.1, a computational component interacts with the outside world by the exchange of messages across interfaces. We distinguish between four interface types of a component, the *Linking Interfaces (LIF)* where the services of the component are offered to the other components of the DAS, the *Configuration and Planning (CP) Interface*, where a component can be configured, a *Diagnostic and Management (DM) Interface*, where the internal elements of a component can be observed and modified for maintenance purposes, and the *local interfaces* to the environment that is controlled by this component.


Diagnostic and Maintenance Interface
Interfaces to local I/O signals
Component
LIF
Configuration and PlanningInterface
Fig. 1: **Interfaces of a Component**

2.2 The LIF Specification

From the point of view of system integration and composability, the LIF is the most important interface of a component. The LIF specification must provide all the information that is needed to understand the behavior and use of the component in the system context. The LIF specification hides the internal implementation details of a component, i.e., *the type of container where the container specific software is executed*. The LIF specification is thus the same for the PIM and the PSM. It should not be of concern to the user of a component whether the component functions are implemented by a program running on a CPU, by an FPGA or by a dedicated hardware machine.

As mentioned before, our LIF is a *message-based interface*. We distinguish between the *operational specification* of the messages and the *meta-level specification* of the messages that pass thru the LIF. The operational specification describes the syntactic structure of the messages and contains the rules how to generate *information chunks* out of the incoming bit stream. Additionally, the timing of the messages, the pre- and post-conditions of the messages, and the sequencing of the messages are part of the operational specification. The timing of state messages is defined by the periodic send instants w.r.t. to a global time base. The planned timing of an event message (production and consumption rate) and the bandwidth of the channel determine the queue spaces that must be allocated to the event message at the sender and receiver. These message queue lengths are monitored at run time in order to detect timing failures. The pre- and post-conditions of the messages establish criteria, whether the contents of a message are in approximate agreement with the semantics of the application. They delimit the state-space of messages to reasonable values. In the SoC architecture these pre- and post-conditions are continually monitored by a diagnostic core in order to record transient failures of a core. Adherence to the operational specifications establishes the interoperability among components.

The *meta-level specification* of the LIF assigns meaning to the information chunks generated at the operational level. The meta-level specification can be expressed in the form of a *state-based* interface model that captures the behavior of the component in the envisioned application context. We call the state of the interface model, as seen from the LIF, the *interface state*. The interface state captures that part of the history of the component that is relevant for its future behavior. The interface state should be made available at the LIF by encoding it in a *periodic interface state message*. This interface-state message can be sent periodically to the diagnostic core of the SoC in order to check the health state of the component and detect any anomalous behavior. A more detailed description of the LIF specification can be found in [6].

Every LIF contains one or more ports that are the entry points (input) or exit points (output) for the messages that cross the LIF. Every port can only handle one message at an instant. If concurrency is desired, multiple ports must be introduced. Dedicated communication channels with known temporal properties connect the ports of the different components of a DAS.

The PIM is an executable model that allows the designer at an early stage of development to check the logic of an application without a concern for all constraints of the target environment – such as memory constraints, power consumption, redundancy strategy, and, above all, cost constraints that are always present in a mass-market product such as an automobile. In a second design phase each PIM has to be transformed to its platform specific representation, the platform specific model (PSM). Each component in the PSM consists of a PSM job and a PSM container. The PSM container is a concrete machine for the execution of the PSM job on a specific hardware/software platform.

Consider for example, a PIM-job that is modeled as a *SCADE Drive* model [7] and then transformed into C using automatic code generation. The compiled C code is the PSM-job and the underlying hardware/software platform (e.g., OSEK operating system, PowerPC processor) is the PSM container. The LIF, however, is the same in the PIM and PSM.

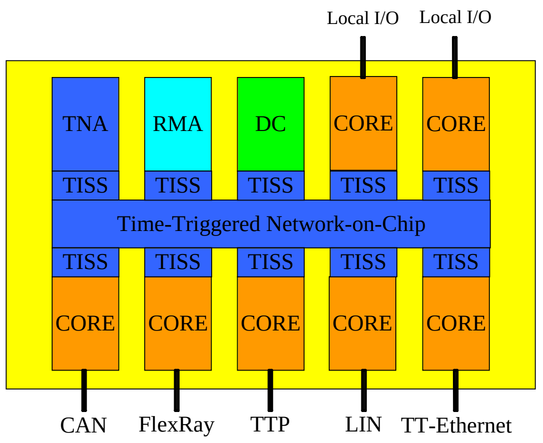


Fig. 2: Overall SoC Architecture

3. The Multi-Core System-on-a-Chip (SoC)

In this Section the architecture of the proposed multi-core System-on-a-Chip (SoC) is described.

3.1 Overall Architecture

The overall architecture of the SoC that forms the execution environment for jobs of the DASes is depicted in Fig. 2. The SoC of Fig. 2 contains *ten cores* that are connected by a time-triggered network on chip (NoC). The cores can be homogenous or heterogeneous. Each core can be a self-contained computer with CPU, memory, and I/O interfaces to the NoC and to its environment. Alternatively, a core could be an FPGA fabric. Each core is an island of synchronicity, i.e., different cores can operate at different frequencies, which can be changed at run-time in order to effect dynamic power management.

There are three special cores depicted in Fig. 2, the trusted network authority (TNA), the resource management authority (RMA) and the Diagnostic Core (DC). The TNA and RMA perform resource management in combination, as outlined in Section 3.3. The Diagnostic Core observes the relevant behavior of all cores in order to detect out-of-norm conditions caused by transient faults.

A core accesses the time-triggered network-on-chip (NoC) via the ports of a so-called *trusted interface subsystem (TISS)*. Using the TISS, the time-triggered SoC architecture provides a dedicated architectural element that protects the access to the time-triggered NoC. Each TISS contains a table which stores a priori knowledge concerning the global points in time of all message receptions and transmissions of the respective core. Since the table cannot be modified by the core, a design fault or a hardware fault restricted to a core cannot affect the exchange of messages by other cores.

The other *seven cores* of the SoC of Fig. 2 host *application jobs* or *gateway jobs*. Each application job of a PIM is assigned to a single core of the SoC. The PIM of an application job has to be transformed to a PSM before it can be executed on the selected core. There are two types of gateway jobs, *internal gateway* jobs and *external gateway* jobs. *Internal gateway* jobs have access to two different DASes of the SoC and support the controlled information exchange between them. External gateway jobs establish a link between the SoC and an external network, such as TTP, FlexRay, or TT Ethernet.

3.2 The Time-Triggered Network-on-Chip

The time-triggered NoC provides dedicated *cut-through multicast communication channels* between the ports of the TISSes of the cores. There is no temporal dependency, neither direct nor indirect, among these communication channels such that the composability of a set of real-time applications can be guaranteed. Each port of a TISS is assigned a conflict-free periodic *send instant* by the TNA. Since the send instants of all TISSes are triggered by the progression of the global time and are guaranteed to be free of contention, there is *no need for arbitration* nor of storage elements within the NoC. The elimination of *arbitration* and *message stores* within the NoC saves valuable real-time and energy.

In order to simplify the communication schedules, all send periods in the NoC are *harmonic fractions* of the physical second. The network clock, which is normally different from the clocks of the cores, can be synchronized with an external time standard, e.g., with GPS time, over an external gateway component (e.g, TT Ethernet [8] in Fig. 2). This network time is made available to all cores of the SoC in order to be able to realize *temporally consistent actions* across the total system. Whenever the network time agrees with the send instant of a port in a TISS, the port will start transmitting its message.

The NoC can transport two different message types, *periodic time-triggered messages* and *sporadic time-triggered messages*. A periodic time-triggered message is sent in every period. It is handled according to state semantics *without any queues*, i.e., a new version overwrites the previous version and the *read operation* is non-consuming. *Sporadic time-triggered messages* are only sent when the sender has produced a new version of the message during the last period. They are queued at the sender until they can be sent and queued at the receiver until they are consumed. Sporadic time-triggered messages provide a flow-controlled event channel with *exactly once semantics* between the respective ports. The bandwidth parameters of this event channel, such as period and message length are determined at design time and enforced by the TNA. The message length, as seen by the user of a TISS, is only limited by the storage area available at the end points of the transmission, since each transmission is implemented as a *pulsed data stream* [9].

3.3 The TNA and RMA

The combination of the trusted network authority (TNA) and the resource management authority (RMA) perform dynamic resource allocation and reconfiguration within the SoC. The following resources are controlled dynamically within the SoC: job assignment to cores, network bandwidth assignment, logical to physical port name translation, and power usage. The TNA is a trusted subsystem that has to be certified to the highest criticality level of any job in the SoC. All resource management decisions are checked and executed by the TNA. The TNA checks assure that no integrity constraint is violated and that all safety-critical jobs receive all necessary resources to perform their mission. The actual scheduling calculations are performed by the Resource Management Authority (RMA). The separation of the resource management into a trusted and un-trusted part has been motivated by certification concerns. In order to keep the trusted part as simple as possible, the complex schedule calculation has been outsourced to a separate core, the RMA. Only the much simpler integrity checks have to be performed by the trusted TNA.

The TNA assigns the *send instants* to the ports of the TISSes of the other cores. A core can read its send instant but is forbidden by hardware means to modify this send instant.

3.4 The Diagnostic Core

The diagnostic core receives all significant output messages and the *internal state messages* of the other cores (consider that the NoC supports multicast transmission) and checks the *post-conditions* that are contained in the LIF specification. Any abnormal behavior of a core, possibly caused by a transient fault, is thus recorded by the diagnostic core and transmitted to a maintenance component via a dedicated diagnostic DAS. The diagnostic architecture of the time-triggered SoC is described in more detail in [10].

4. DAS Integration

As explained in Section two, each DAS is developed in isolation. At the PIM level, a DAS consists of a set of jobs (including their containers) that communicate via dedicated communication channels. During the integration the jobs from different DASes must be mapped to the cores of a single SoC. A core will host exactly one job of a DAS.

4.1 PSM Generation

In a first step the PIM of a job must be transformed to generate the appropriate platform-specific model (PSM). The PSM depends on the hardware capability of the SoC core that will host the job. The formal model of the PIM has to be transformed into a representation that can be executed on the target core. If required, appropriate middleware and operating system modules must be added in order to generate the PSM for the selected target core.

In particular, the middleware realizes specific virtual overlay networks that may be required by legacy jobs. For example, if a legacy job uses the CAN protocol for communication, the middleware must emulate a dedicated CAN on an encapsulated communication channel of the time-triggered NoC. In the context of the DECOS project [11] we have shown that an emulated CAN has the same temporal properties as a physical CAN network and the application does not notice any difference between the emulated and the physical CAN.

4.2 Integration of Legacy Software

The proposed architecture supports the integration of available legacy software. The CPU of an existing ECU can be selected as the core of the multi-core SoC. This CPU can access its local process I/O signals with the existing legacy software without any modification. No change of the application software is thus needed. It is only required to modify the middleware in order to emulate the services of the legacy platform on the NoC.

The communication between different subsystems is realized by a gateway. A gateway can send and receive messages from two or more different subsystems. We distinguish between two types gateways, *intra-DAS* gateways and *inter-DAS* gateways.

An *intra-DAS gateway* connects two different communication systems of a single DAS by an overlay network. Such a gateway is transparent to the application software. For example, if two jobs of a legacy software system are used to communicate via the CAN protocol, but the physical connection is realized by a concatenation of an on-chip time-triggered NoC and an off-chip TT Ethernet, the middleware and the gateways can realize an overlay CAN network that has the same naming structure and temporal properties as the physical CAN used in the legacy system. A number of different fully encapsulated overlay CAN networks can be implemented on a single physical time-triggered network that supports an appropriate bandwidth. For example, the NoC can support hundreds of different fully encapsulated CAN channels. If the off-chip communication is realized by a 100 Mbit/sec TT Ethernet, about one percent of the bandwidth is sufficient to implement an overlay CAN channel. Even if an existing network – either the NoC or an off-chip network – is changed and replaced by a new one the legacy application software does not need to be modified. The proposed system architecture thus preserves the existing investment in the application software, even in the face of technology obsolescence of the hardware.

Inter-DAS gateways support the interconnection of two different DASes. It must be decided at the application level which information from one DAS should be made available to the other DAS and what constitutes correct and erroneous behavior of a partner DAS.

5. Fault Isolation and Error Containment

Because of the architecture-inherent error-detection mechanisms the encapsulation – and as a consequence the composability [12] – of the DASes in this integrated architecture is superior to the error containment that is achieved in a federated architecture, where different DASes exchange messages via a common CAN bus. In the first part of this Section we analyze the fault isolation and error-containment properties of the proposed architecture with respect to design (software) faults. The handling of physical faults is treated in the subsequent part.

5.1 Design Faults

We make the assumption that the basic hardware and the trusted network authority (TNA) are free of design faults. Since the TNA software is generic and part of every SoC, extensive verification efforts are justified to substantiate this assumption.

Since a core is a computer with its own CPU and memory that is not shared among multiple jobs, a software fault in the PSM of a job can have an immediate effect on this job only. A core is thus an encapsulated fault-containment unit. The effect of a fault in a core – the emerging error – can propagate to other cores by the transmission of erroneous messages. A message can be erroneous with respect to the *value domain* or to the *time domain*. Timing errors of messages are detected and covered by the TISS, since the *send instant* of messages cannot be modified by the software of a core, but only by the TNA. Since every message is automatically sent to the diagnostic core, there is high probability that the diagnostic core will detect an erroneous value in a message by executing the *post condition* that is part of the LIF specification.

5.2 Physical (Hardware) Faults

We distinguish between four types of hardware faults, as characterized in Table 1.

Table 1: Hardware Fault Characterization

Fault Extent	Fault Persistence
Local to a core	Transient
Global	Permanent

Considering available field data [13] transient faults are by orders of magnitude more probable than permanent faults. According to the literature [14] radiation induced high energy ions are expected to be by far the most important cause of transient faults in future advanced computer systems, such as our SoC. Single event upsets (SEU) which are caused by ambient radiation are temporary electric disturbances that effect a small area of a chip – in the order of a few μm^2 – that may provoke one or more bit flips in the affected area. Since only the stored information, the *state*, but not the hardware of the chip is permanently destroyed, a transient error can be repaired by a state recovery action. Considering the physical dimensions of a core on our SoC it can be expected that an SEU will impact a single core of the multi-core SoC only. The application software must thus contain mechanisms for the recovery of single cores. For example, it might be advantageous to save the state of one core periodically in a different core of the SoC. In high dependability applications, systematic state recovery can be implemented by on-chip triple modular redundancy (TMR). In safety-critical applications off-chip TMR must be implemented, since in these applications the probability of global chip-wide faults cannot be tolerated.

5.3 Experimental Evaluation

In the context of the DECOS project we have implemented a prototype of the proposed multi-core SoC in order to experimentally evaluate the claimed *fault isolation* and *error containment properties* of the proposed time-triggered SoC. Therefore, the prototype setup contains a core performing fault injection to simulate arbitrary value and timing failures.

Each component of the SoC consists of two single board computers, one (a Soekris Engineering net 4521) implementing a *core* and the second one (a Soekris Engineering net4801) implementing a TISS. As an operating system, both single board computers use the realtime Linux variant Real-Time Application Interface (RTAI) [15].

A 100 Mbit/sec time-triggered (TT) Ethernet simulates the time-triggered network-on-chip. A monitoring device has been connected to the TT-Ethernet switch and has used its Ethernet interface in promiscuous mode. The tool *WireShark* has been used to observe and log the traffic of the entire TTE network. The analysis of the collected logs has yielded the following results:

• ***No discontinuities:*** In the experiments, all sent messages have included sequence numbers. The logs have indicated that no messages have been lost due to the behavior of the faulty core.

• ***No additional messages:*** Other than the messages configured by the TNA, no messages have been observed on the TTE network.

• ***No effect on temporal properties:*** The temporal properties (i.e., bandwidth, latency, message order) of the TTE network with fault injection by a core have been identical to the temporal properties without fault injection.

In the mean time we have received funding to implement the TT-NoC on an FPGA, which will give us a test-bed that is in better agreement with the envisioned NoC.

6. Conclusion

Many technical and economic advantages can be accrued if the number of ECUs and the number of wiring points within the electronic system aboard a car are reduced. For this purpose, it is necessary to integrate multiple functions on a single ECU. The critical issue to achieve this challenging goal is the composability and error-containment capability of the chosen system architecture. In our opinion Systems-on-a-Chip that host multiple cores connected by a time-triggered Network-on-Chip provide the required composability and error containment to achieve this integration.

In this paper we have described a model-based development method to program these multi-core SoCs and have presented some experimental data to demonstrate that the required goals of error containment can be achieved.

The proposed SoCs form the ideal hardware base for realizing the integration objectives of AUTOSAR. As part of future work, we plan to integrate the proposed SoC architecture into the AUTOSAR framework.

Acknowledgement

This work has been supported, in part, by the European IST project DECOS under project No. IST-511764.

References

[1] J. Leohold, "Architecture of Automotive Systems," in *Summer School on Architectural Paradigms for Dependable Embedded Systems*, TU Vienna, Austria, 2005.

[2] R. Hammett, "Flight-Critical Distributed Systems--Design Considerations," 2002, pp. 13-B3.1-13B3.5.

[3] ARINC, "ARINC Specification 651: Design Guide for Integrated Modular Avionics," Aeronautical Radio, Inc., 2551 Riva Road, Annapolis,Maryland 21401 1991.

[4] AUTOSAR, "AUTOSAR - Technical Overview V2.0.1," AUTOSAR GbR 2006.

[5] M. Gruber, editor, "The IST IP DECOS Project Proposal," Austrian Research Center, Vienna, Austria 2004.

[6] H. Kopetz and N. Suri, "Compositional Design of Real-Time System: A Conceptual Basis for the Specification of Linking Interfaces," in *ISORC 2003--The 6th International Symposium on Object Oriented Real-Time Computing*, Hakodate, Japan, 2003, pp. 51-60.

[7] B. Dion, "Efficient Development of Embedded Automotive Software with IEC 61508 Objectives using SCADE Drive," *VDI-Gesellschaft Fahrzeug- und Verkehrstechnik*, 2005.

[8] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer, "The Design of TT Ethernet," in *ISORC 2005*, Seattle, 2005.

[9] H. Kopetz, "Pulsed Data Streams," Institut für Technische Informatik, TU Wien, Austria 2006.

[10] C. El-Salloum, R. Obermaisser, B. Huber, and H. Kopetz, "A Time-Triggered System-on-a-Chip Architecture with Integrated Support for Diagnosis," in *Workshop on Diagnostic Services in Network-on-Chips* Nice, France, 2007.

[11] R. Obermaisser, "Reuse of CAN-based Legacy Applications in Time-Triggered Architectures," *IEEE Transactions on Industrial Informatics,* vol. 2, pp. 255-268, 2006.

[12] H. Kopetz, "Composability in the Time-Triggered Architecture," in *SAE World Congress*, Detroit, USA, 2000, pp. 1-8.

[13] P. Peti, R. Obermaisser, A. Ademai, and H. Kopetz, "A Maintenance-Oriented Fault-Model for the DECOS Integrated Diagnostic Architecture," in *Workshop on Parallel and Distributed Real-Time Systems*, 2005.

[14] R. Baumann, "Soft Errors in Advanced Computer Sytems," *IEEE Design and Test of Computers,* pp. 258-266, 2005.

[15] D. Beal, E.Bianchi, L. Dozio, S. Hughes, P.Mantegazza, and S. Papacharalambous, "RTAI: Real-Time Application Interface," *Linux Journal*, 2000.