

HABILITATIONSSCHRIFT

# **A Time-Triggered Integrated Architecture**

eingereicht an der Technischen Universität Wien

Fakultät für Informatik

von

Dipl.-Ing.Dr.techn.Roman Obermaisser

zur Erlangung der Lehrbefugnis

für das Fach Technische Informatik

Wien, im September 2008



# Contents

Preface	vii
1 Paradigm of Time-Triggered Control	viii
2 The Scope of Time-Triggered Integrated Architectures	x
3 Overview of Selected Papers	xii
References	xix
An Integrated Architecture for Future Car Generations	1
<i>Roman Obermaisser, Philipp Peti, Fulvio Tagliabo</i>	
1 Introduction	1
2 Federated vs. Integrated Architectures	3
3 The DECOS Integrated Architecture	4
4 Today's Automotive Architectures	7
5 An Architecture for Future Car Generations	9
6 Benefits of the Proposed Architecture	18
7 Related Work	21
8 Conclusion	25
References	26
Temporal Partitioning of Communication Resources in an Integrated Architecture	29
<i>Roman Obermaisser</i>	
1 Introduction	29
2 Basic Concepts and Related Work	32
3 System Model	34
4 Virtual Networks	35
5 Experimental Setup with Prototype Implementation of Virtual Networks	41
6 Hypotheses and Experiments	43
7 Results	48
8 Discussion of Results	51
9 Conclusion	52
References	53
Reuse of CAN-based Legacy Applications in Time-Triggered Architectures	57
<i>Roman Obermaisser</i>	
1 Introduction	57
2 Related Work of Integration of ET and TT Communication	59
3 Controller Area Network Emulation	61
4 Event Service	63
5 Protocol Emulation	67
6 Front-End	72
7 Results and Discussion	74
8 Conclusion	79

References	79
Detection of Out-of-Norm Behaviors in Event-Triggered Virtual Networks	83
<i>Roman Obermaisser, Philipp Peti</i>	
1 Introduction	83
2 DECOS Architecture	85
3 Detection of Out-of-Norm Behavior	85
4 Exploitation of Out-of-Norm Behavior Detections	90
5 Implementation	92
6 Discussion	93
References	93
A Model-Driven Framework for the Generation of Gateways in Distributed Real-Time Systems	97
<i>Roman Obermaisser</i>	
1 Introduction	97
2 Related Work	99
3 The DECOS Integrated Architecture	101
4 Gateways based on a Real-Time Database	103
5 Gateway Specification Model	106
6 Implementation	110
7 Example	114
8 Discussion	119
References	119
A Transient-Resilient System-on-a-Chip Architecture with Support for On-Chip and Off-Chip TMR	121
<i>Roman Obermaisser, Hubert Kraut, Christian Salloum</i>	
1 Introduction	121
2 Time-Triggered System-on-a-Chip Architecture	124
3 Fault Tolerance	128
4 Reliability Modeling	133
5 Results	137
6 Conclusion	138
7 Acknowledgments	139
References	139

## Introduction

The progress of the semiconductor industry makes it economically feasible to replace mechanical and hydraulic control systems by embedded computer systems and to increase their functionality far beyond the capabilities of the systems they replace. Embedded computer systems have lead to dramatic improvements with respect to functionality, safety, and cost in many domains (e.g., automotive systems, avionic systems, industrial control, medical systems, or consumer applications).

For instance, the automotive industry shares the view that in the next 10 years 90% of its expected innovations will be based on novel IT applications within the car and its environment [Rep05, p. 32]. Vision-based driver assistance systems are an example of such an innovation. By guiding the driver in routine traffic scenarios and assisting the driver in critical situations, these systems are expected to significantly reduce the number and impact of traffic accidents [LS04]. In the aerospace industry the deployment of fly-by-wire systems has led to a safer and more dependable air-transport system [Col99].

Large and complex electronic systems, such as those deployed in automotive and avionic applications, are typically structured into multiple application subsystems each providing a specific application service and exhibiting a particular criticality level. Traditionally, these application subsystem (e.g., multimedia, comfort, powertrain, safety in a car) were constructed based on *federated architectures* [Rus01]. In federated architectures, each application subsystem is implemented using a dedicated distributed computer system with its own computing nodes and networking infrastructure.

Through this physical segregation of application subsystems, federated architectures facilitate the establishment of essential system properties such as composability [Sif05], modular certifiability [Rus01b], fault containment, and error containment [LH94]. However, federated architectures in conjunction with the trend of pervasive embedded systems have lead to the unintended side-effect of a dramatic increase of the number of embedded computing nodes. For example, present day premium cars can contain up to 100 electronic control units [LSPS04]. Likewise, federated architectures in the avionic domain have caused significant weight and cost due to large numbers of line-replaceable units with hundreds of pounds of excess weight [Buc05, p. 22-10].

A strategy to limit the growth of computing nodes and networks is the integration of multiple application subsystems into a single *integrated architecture*. This strategy is proposed by several system architectures such as Integrated Modular Avionics (IMA) [Aer91], Automotive Open System Architecture (AUTOSAR) [AUT06b], or Dependable Embedded Components and Systems (DECOS) [OPT07].

However, one of the key obstacles to the integration of multiple application subsystems of differing criticality into a single integrated architecture is the potential increase in the complexity of the integrated system through unintended interference between application subsystems via the

shared platform. In order not to increase the complexity in an integrated architecture, it has to be ensured that the activities of one application subsystems cannot interfere with the activities of other application subsystems. Otherwise, the properties of an application subsystems will depend on the behavior of the other application subsystems. In such a case, the analysis of an application subsystems becomes a global problem in which all potential interactions between application subsystems need to be considered in addition to the activities within the application subsystems. These potential interactions, however, do not stem from the application, but introduce an additional accidental complexity beyond the inherent complexity of the application.

Consider for example the integration of multiple distributed application subsystems using a Controller Area Network (CAN) [Bos91]. Even if the complexity of each single application subsystem is low, the analysis of the integrated system becomes more difficult due to the emergent complexity induced by the hidden interactions between the application subsystems. For example, high priority CAN message of one application subsystem can delay lower priority CAN messages of the other application subsystems. Thus, the temporal behavior of one application subsystem depends on the communication activities of the other application subsystems on the CAN bus.

In addition to the behavior during normal operation, unintended interactions in the presence of faults need to be considered. In the example of the CAN bus, a hardware fault that is not covered by the error counters or a design fault can lead to error propagation between application subsystems.

A federated system trivially rules out unintended interference between application subsystems by providing physically-separated computational resources (e.g., CPU, memory) and communication resources (e.g., networks). The integrated architecture, on the other hand, needs to impose fault and error containment in order to prevent accidental complexity in the presence of faults. The challenge is to guarantee that faults do not affect or damage other, working parts of the system or restrict services provided by other, working parts of the system (e.g., by unrestricted usage of resources). The faults that need to be considered include transient and permanent hardware faults, design faults, imprecise specifications, and accidental operational faults.

This thesis describes the services of an integrated architecture that meets this challenge. In order to enable the development of embedded applications with multiple subsystems and criticality levels, this thesis systematically addresses the required architectural services such as communication services, gateway services, diagnostic services, legacy integration services, and fault-tolerance services. The proposed integrated architecture is based on the time-triggered control paradigm, since the consequent allocation of resources based on the progression of time facilitates the encapsulation and fault isolation of application subsystems as motivated above. Therefore, the architecture is called a *time-triggered integrated architecture*.

The following sections give a brief introduction to the basic concepts of time-triggered control and elaborate on the services that are required to provide a foundation for the integration of application subsystem on a common architecture. This introduction is followed by an overview of the papers that are included in this habilitation thesis. The overview discusses the relevance of each paper and shows how each paper relates to the problem domains of the time-triggered integrated architecture.

## 1. Paradigm of Time-Triggered Control

Event-triggered and time-triggered communication systems are two different paradigms for the construction of the communication infrastructure of a distributed real-time system. The major difference between these paradigms lies in the location of control. Event-triggered communication

systems are based on external control via event triggers. The decision when a message is to be transmitted is within the sphere of control of the application in the host. In a time-triggered communication system, the communication controller decides autonomously about the global points in time at which messages are transmitted. This autonomous control of a time-triggered communication system results from restricting control signals to time triggers, which are independent of the state changes in the environment and the host computer.

## Supported Information Semantics

A time-triggered communication system is designed for the periodic exchange of messages carrying state information. These messages are called *state messages*. The self-contained nature and idempotence of state messages eases the establishment of state synchronization, which does not depend on exactly-once processing guarantees. Since applications are often only interested in the most recent value of a real-time object, old state values can be overwritten with newer state values. Hence, a time-triggered communication system does not require message queues.

## Communication Network Interface

As depicted in Figure 1, the Communication Network Interface (CNI) of a time-triggered communication system acts as a temporal firewall [Kop01]. The sender can deposit information into the CNI according to the information push paradigm, while the receiver must pull information out of the CNI. A time-triggered transport protocol autonomously carries the state information from the CNI of the sender to the CNIs of the receivers. Since no control signals cross the CNI, temporal fault propagation is prevented by design.

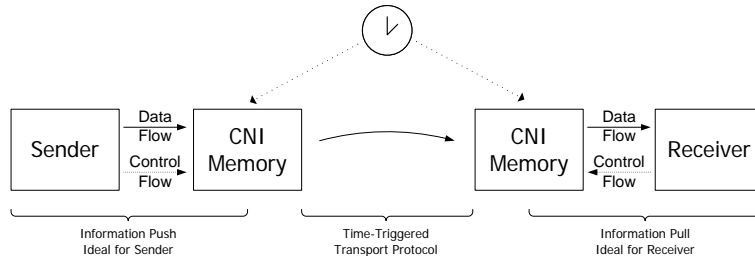


Figure 1. Data Flow (Full Line) and Control Flow (Dashed Line) of a Temporal Firewall Interface [Kop01]

The state messages in the CNI memory form two groups. Those state messages that are written by the host application represent the node's service providing linking interface (SPLIF). The communication controller reads these messages and disseminates them during the slots reserved for the node via the underlying TDMA scheme. Those messages that form the node's service requesting linking interface (SRLIF) are written by the communication controller and read by the host application.

Consistency of information exchanged via the CNI can be ensured by exploiting the a priori knowledge about the points in time when the communication system reads and writes data into the CNI. The host application performs implicit synchronization by establishing a phase alignment between its own CNI accesses and the CNI accesses of the communication controller. A different approach is the use of a synchronization protocol, such as the *Non-Blocking Write Protocol* [KR93].

## Transport Protocol

The media access control strategy of a time-triggered communication system is Time-Division Multiple Access (TDMA). Time-Division Multiple Access (TDMA) statically divides the channel capacity into a number of slots and assigns a unique slot to every node. The communication activities of every node are controlled by a time-triggered communication schedule. The schedule specifies the temporal pattern of message transmissions, i.e., at what points in time nodes send and receive messages. A sequence of sending slots, which allows every node in an ensemble of  $n$  nodes to send exactly once, is called a TDMA round. The sequence of the different TDMA rounds forms the cluster cycle and determines the periodicity of the time-triggered communication.

The a priori knowledge about the times of message exchanges enables the communication system to operate autonomously. The temporal control of communication activities is within the sphere of control of the communication system. Hence, the correct temporal behavior of the communication system is independent of temporal behavior of the application software in the host computer and can be established in isolation.

## Flow Control

Time-triggered communication systems employ *implicit flow control* [Kop97]. Sender and receiver agree a priori on the global points in times when messages are exchanged. Based on this knowledge, a component's ability for handling received messages can be ensured at design time, i.e., without acknowledgment messages. Implicit flow control is well-suited for multicast communication relationships, because a unidirectional data flow involves only a unidirectional control flow (*elementary interface* [Kop99]). Multicasting is required in many applications, e.g., in control loops or for the realization of active redundancy in order to provide the same inputs to replicated components.

## 2. The Scope of Time-Triggered Integrated Architectures

A technical system architecture is a *framework for the construction of a system for a chosen application domain that provides generic architectural services and imposes an architectural style* [Kop06, p. 34]. The *architectural style* consists of rules and guidelines for the partitioning of a system into subsystems and for the design of the interactions among subsystems. The purpose of the architectural style is to constrain developers in such a way that the ensuing system exhibits certain properties, such as composability, dependability or modular certifiability. The *generic architectural services* provide to system designers a validated stable baseline for the development of applications. The architectural services solve recurring problems of many actual systems "once and for all".

The following architectural services are essential in order to support the development of distributed embedded real-time systems integrating multiple application subsystems with different levels of criticality and different requirements concerning the underlying platform:

- **Service 1 – Communication service.** The purpose of the communication services is to provide the infrastructure for integrating components to the overall system. In order to address the complexity management challenge, the communication services have to support the principle of *non-interfering interactions* [KO02]. Given disjoint subgroups of cooperating components, this principle states that the use of the communication services by one subgroup may not interfere with the communication services provided to the other subgroups (including non functional properties such as temporal and dependability constraints). If this



principle is not satisfied, then the integration within one component-subgroup depends on the proper behavior of the other component-subgroups. Thus, the system complexity would increase by requiring a global analysis to reason about the behavior of the component-subgroups.

Furthermore, the communication services of the time-triggered integrated architecture also need to support non-interference in the presence of faults. For this purpose, the system must be structured into independent fault containment regions and the communication services have to incorporate mechanisms for temporal and spatial partitioning [Rus99]. Thereby, the time-triggered integrated architecture ensures that the consequences of a component error do not affect the behavior of any other component that does not rely on the services of the erroneous component.

Based on these fault containment capabilities, the communication services must enable the establishment of error containment in order to prevent the propagation of fault effects manifested as erroneous data across the boundaries of fault containment regions. An important mechanism for error containment is active redundancy through Triple Modular Redundancy (TMR) (cf. service 5), which requires deterministic communication services [Pol96] as a prerequisite for exact voting.

- **Service 2 – Gateway service.** When integrating multiple application subsystem in a common architecture, information exchanges between these application subsystems are essential to realize composite services that involve more than one application subsystem and to reduce redundant computations and sensors. A major challenge is to resolve the property mismatches [C. 02] at the interfaces between these application subsystems, such as incoherent naming, divergent syntax, or different communication protocols. Secondly, fault isolation capabilities are required to prevent common mode failures induced by the propagation of faults between application subsystems. A gateway service provides architectural support for handling these two challenges. The purpose of a gateway service is the selective redirection of information in conjunction with the necessary property transformations and error containment mechanisms.
- **Service 3 – Legacy platform services and Application Programming Interfaces (APIs).** When introducing a new system architecture, backward compatibility is essential in order to protect the investments in legacy applications. In addition to the high cost of reimplementing legacy applications, the missing experience with new technologies can introduce an additional risk when replacing well-tested legacy applications. The time-triggered integrated architecture supports this requirement by providing the platform services and APIs of the relevant legacy platforms. For example, when reusing automotive legacy applications the provision of the communication services of the CAN protocol [Int93] and CAN-based APIs [Vol03, CAN05] is essential since CAN is the most widely used communication protocol in today's distributed automotive computer systems.
- **Service 4 – Diagnostic service.** A further class of services that are important in a time-triggered integrated architecture are diagnostic services. A diagnostic service monitors the functionality and performance of components and subsystems. Thereby, faulty components can be identified in order to perform online error recovery actions (called active diagnosis) or off-line maintenance (called passive diagnosis).

Ideally, maintenance oriented diagnostic services should include functions for recording and analyzing every anomaly in the system in order to gain an effective assessment of component conditions. Diagnosis should consider anomalies, even if they are masked by fault-tolerance mechanisms and therefore rendered transparent to applications. For example, a triple-modular-redundancy system can provide information about a disagreeing processor in addition to masking the effects of faults.

A fundamental prerequisite for a maintenance oriented diagnostic service are the error containment capabilities of the architecture, because the absence of error containment mechanisms would cause the inability to identify the locations of failures. The resulting trouble-not-identified phenomenon [TAP02] is currently a prevalent problem, as it leads to the replacement of correct components (e.g., in the automotive industry).

In case of active diagnosis, on the other hand, the diagnostic information is used to achieve fault-tolerance by directly intervening in the system behaviour by means of reconfiguration (e.g., migration of services to spare components, graceful degradation). Since for safety-critical systems, active diagnosis is directly related to safety properties, the diagnostic subsystem must be trusted and certified to the highest criticality level in the given application.

- **Service 5 – Redundancy management service.** An application service can be implemented by a group of redundant, independent components in order to ensure that the application service remains available despite the occurrence of component failures. If the number and types of component failures are covered by the underlying failure mode assumptions, the group will mask failures of its members [Cri91].

A common approach for masking component failures is N-modular redundancy (NMR) [Avi75, LA90]. N replicas receive the same requests and provide the same service. The output of all replicas is provided to a voting mechanism, which selects one of the results (e.g., based on majority) or transforms the results to a single one (average voter). The most frequently used N-modular configuration is triple-modular redundancy (TMR). By employing three components and voters, a single consistent value failure in one of the constituting components can be tolerated.

The major strength of group masking is the ability to handle component failures systematically at the architecture level, i.e. transparently to the application. The purpose of the redundancy management services of the architecture is the provision of mechanisms required for managing the redundant groups of replicas in a way that masks component failures and makes the group functionally indistinguishable from a single replica.

### 3. Overview of Selected Papers

The following sections give a short overview of the papers that are included in this thesis. For each paper, the research contribution and the relevance to the problem domains of time-triggered integrated architectures are stated.

#### An Integrated Architecture for Future Car Generations

*Roman Obermaisser, Philipp Peti, Fulvio Tagliabo. Real-Time Systems Journal, Volume 36, 2007, pages 101–133, Springer.*

**Summary.** This paper provides an architecture overview of the time-triggered integrated system architecture. At a logical level, systems are modeled as Distributed Application Subsystems (DASs) each consisting of one or more jobs that interact through the timely exchange of messages. At a physical level, systems consist of node computers interconnected by a time-triggered network. Application developers are provided with architectural services as a stable baseline for the development of the applications, e.g., communication services, gateway services, and diagnostic services. Existing embedded systems can be mapped to the integrated architecture by replacing node computers of the federated system with jobs that can share node computers in the integrated architecture. This process is exemplified for the E/E system of a present day car.

**Research Contribution.** The research contributions include (1) the systematic exploration of the advantages and disadvantages of integrated system architectures, (2) the definition of structuring rules at logical and physical levels, (3) the identification and conceptualization of architectural services, and (4) a mapping from today's automotive systems to the integrated architecture.

**Relevance for the Problem Domain of Time-Triggered Integrated Architectures.** The structuring rules are required as a basis for the design of applications, as well as for the definition of the architectural services and the formulation of the fault assumptions. In order to move to the time-triggered integrated architecture, architectural services are required as a foundation for the realization of applications. The paper identifies and conceptualizes these services, which will be addressed in detail in the following papers of this thesis. Since existing applications (e.g., in the automotive domain) are built according to a different architectural paradigm, it is necessary to explain how the underlying architecture in these applications can be replaced. The paper gives such a migration path for the automotive domain.

## Temporal Partitioning of Communication Resources in an Integrated Architecture

*Roman Obermaisser. IEEE Transactions on Dependable and Secure Computing, October 2007, IEEE Computer Society.*

**Summary.** This paper focuses on the communication services of the time-triggered integrated architecture. The communication services are provided by virtual networks on top of an underlying time-triggered physical network. Virtual networks exhibit predefined temporal properties (i.e., bandwidth, latency, latency jitter). Due to temporal partitioning, the temporal properties of messages sent by a job are independent from the behavior of other jobs, in particular from those within other DASs. Partitioning facilitates the management of complexity and prevents the propagation of faults, in particular between components of different criticality levels. The paper presents the mechanisms for temporal partitioning and gives experimental evidence. Rigid temporal partitioning is achievable, while at the same time meeting the performance requirements imposed by present-day automotive applications and those envisioned for the future (e.g., X-by-wire). For this purpose, an experimental framework with an implementation of virtual networks on top of a TDMA-controlled Ethernet network is used.

**Research Contribution.** The main research contributions of this paper include (1) the mechanisms for temporal partitioning in the communication system of an integrated architecture, (2) the

definition of a communication infrastructure for heterogeneous DASs (e.g., with different criticalities), and (3) the experimental assessment of partitioning and performance.

**Relevance for the Problem Domain of Time-Triggered Integrated Architectures.** The relevance of this paper for the problem domain of the thesis lies in the definition of the communication services of the time-triggered integrated architecture, which provide the infrastructure for the composition of components. The paper introduces this infrastructure by means of virtual networks on top of a time-triggered physical network.

### Reuse of CAN-based Legacy Applications in Time-Triggered Architectures

*Roman Obermaisser. IEEE Transactions on Industrial Informatics, Volume 2, Number 4, 2006, pages 255–268.*

**Summary.** This paper introduces a CAN communication service for the time-triggered integrated architecture. This service provides to CAN-based applications the same interface as in a conventional CAN system, thus minimizing redevelopment efforts for CAN-based legacy software. For this purpose, a CAN emulation middleware operates between a time-triggered operating system and the CAN-based applications. In a first step, the middleware establishes event channels on top of the time-triggered communication network in order to support on-demand transmission requests at a priori unknown points in time. The middleware then emulates the CSMA/CA media access protocol of a physical CAN network for passing messages received via event channels to the application in the correct temporal order. Finally, the API of the widely-used HIS/VectorCAN driver provides a handle-based programming interface with support for message filtering and callbacks. A validation setup with a TTP cluster demonstrates that the CAN emulation can handle CAN-based legacy software and a real-world communication matrix provided by the automotive industry.

**Research Contribution.** The research contributions of this paper include the (1) integration of two communication paradigms (layering of a virtual CAN network for event-triggered communication on top of time-triggered communication), (2) the introduction of an algorithm for emulating the CSMA/CA media access protocol in order to achieve the message ordering of a physical CAN network, and (3) the experimental assessment of the performance including the comparison with a physical CAN network.

**Relevance for the Problem Domain of Time-Triggered Integrated Architectures.** Despite the transition from federated to integrated architectures, it is essential to allow the reuse of existing applications in order to preserve investments and prevent cost for reimplementations. For this purpose the problem domain of time-triggered integrated architectures includes the challenge to emulate existing platforms. This paper provides such an emulation for the most important communication protocol in today's cars, the CAN protocol, which is also used in many other domains such as Avionics, industrial control, or medical equipment.

## Detection of Out-of-Norm Behaviors in Event-Triggered Virtual Networks

*Roman Obermaisser, Philipp Peti. Proceedings of the 5th IEEE International Conference on Industrial Informatics, Volume 1, pages 541–546. Vienna, Austria, July 2007 (best paper award).*

**Summary.** This paper introduces architectural support for diagnosis and shows that the underlying time-triggered network significantly improves the accuracy of the error detection mechanisms in comparison to federated architectures used today. Detectors for out-of-norm behavior are distributed across the nodes of the distributed real-time system. The detectors produce diagnostic messages augmented with information about the location and time of the detection event. Due to the fault isolation and the global time base of the integrated architecture, additional spatial and temporal information (besides the value domain) is available in the diagnostic messages and forms a meaningful input to a subsequent analysis process. The proposed framework manages the inherently imprecise temporal specifications of event-triggered DASs by correlating diagnostic messages along value, space and time. Thereby, a discrimination between a correct behavior of the computer system (e.g., triggered by rare conditions in the environment) and different fault classes (e.g., design faults, physical faults) becomes feasible.

**Research Contribution.** The main research contribution of this paper is the introduction of diagnostic mechanisms that handle imprecise specifications without a sharp line that would allow a classification into correct and incorrect behavior. The concept of out-of-norm behavior is introduced in order to capture improbable behavior that probabilistically represents a failure. Out-of-norm behavior is detected using timed automata that encode checks in the value and temporal domain for the detection of those interface states that bear the potential of revealing job faults. An accurate assessment of the health state of individual components is achieved through the correlation of out-of-norm in the domains of value, time and space.

**Relevance for the Problem Domain of Time-Triggered Integrated Architectures.** The contribution to the problem domain of time-triggered integrated architectures lies in the definition of diagnostic architectural services for the identification of faulty components. The diagnostic services are an important basis for maintenance activities, engineering feedback, and fault-tolerance mechanisms.

## A Model-Driven Framework for the Generation of Gateways in Distributed Real-Time Systems

*Roman Obermaisser. Proceedings of the 28th IEEE Real-Time Systems Symposium, pages 93–104, Tucson, Arizona, USA, December 2007.*

**Summary.** This paper presents a generic framework for gateways, which enable message exchanges across DAS boundaries in order to exploit redundancy and to coordinate the behavior of DASs. A gateway can connect a virtual network to the virtual networks of other DASs, as well as to physical networks outside the integrated computer system (e.g., a fieldbus or a legacy cluster). We introduce gateways that contain structured collections of time-sensitive variables associated with timing information (called real-time databases) in order to separate the DASs. The proposed

framework includes a code generation tool that produces a middleware layer for forwarding data from a real-time database to the different networked components, as well as for collecting data from networked components in order to update time-sensitive variables in a real-time database. The input of the code generation tool is a formal gateway specification model, which defines the interaction protocol of the networked components using state machines with timing constraints. We demonstrate the feasibility of the proposed gateway framework with an automotive example in a prototype implementation based on a time-triggered communication protocol.

**Research Contribution.** The research contributions of the paper comprise (1) a novel solution for gateways based on a real-time database, (2) a framework for the modular construction of gateways, (3) execution semantics and support for automatic code generation, and the (4) support for real-time systems encompassing multiple DASs.

**Relevance for the Problem Domain of Time-Triggered Integrated Architectures.** Gateways are an important part of the architectural services of a time-triggered integrated architecture. Complex integrated computer systems can encompass multiple DASs, such as a multimedia, a powertrain, a comfort and a safety subsystem in the in-vehicle electronic system of a typical premium car. Information exchanges between these DASs are essential to realize composite services that involve more than one DAS and to reduce redundant computations and sensors. A major challenge is to resolve the property mismatches at the interfaces between DASs, such as incoherent naming, divergent syntax, or different communication protocols. Also, fault isolation capabilities are required to prevent common mode failures induced by the propagation of faults between DASs.

## A Transient-Resilient System-on-a-Chip Architecture with Support for On-Chip and Off-Chip TMR

*Roman Obermaisser, Hubert Kraut, Christian Salloum. Proceedings of the 7th European Dependable Computing Conference, pages 123–134, Kaunas, Lithuania, May 2008.*

**Summary.** The ongoing technological advances in the semiconductor industry make Multi-Processor System-on-a-Chips (MPSoCs) more attractive, because uniprocessor solutions do not scale satisfactorily with increasing transistor counts. In conjunction with the increasing rates of transient faults in logic and memory associated with the continuous reduction of feature sizes, this situation creates the need for novel MPSoC architectures. This paper introduces such an architecture, which supports the integration of multiple, heterogeneous IP cores that are interconnected by a time-triggered Network-on-a-Chip (NoC). Through its inherent fault isolation and determinism, the proposed MPSoC provides the basis for fault tolerance using Triple Modular Redundancy (TMR). On-chip TMR improves the reliability of a MPSoC, e.g., by tolerating a transient fault in one of three replicated IP cores. Off-chip TMR with three MPSoCs can be used in the development of ultra-dependable applications (e.g., X-by-wire), where the reliability requirements exceed the reliability that is achievable using a single MPSoC. The paper quantifies the reliability benefits of the proposed MPSoC architecture by means of reliability modeling. These results demonstrate that the combination of on-chip and off-chip TMR contributes towards building more dependable distributed embedded real-time systems.

**Research Contribution.** The research contributions of the paper are architectural services for the transparent realization of fault-tolerance through Triple Modular Redundancy (TMR). Issues of replica determinism are addressed and the granularity for TMR is raised from logic circuit to IP core level. Furthermore, the complementarity of on-chip and off-chip TMR is quantitatively evaluated through reliability modeling.

**Relevance for the Problem Domain of Time-Triggered Integrated Architectures.** Architectural services for the transparent implementation of fault-tolerance are relevant for the following reasons: Firstly, mechanisms for fault-tolerance are a prerequisite for the construction of ultra-reliable computer systems in safety-critical applications with the requirement of a maximum failure rate of  $10^{-9}$  critical failures per hour. Today's technology does not support the manufacturing of electronic devices with failure rates low enough to meet these reliability requirements. Since component failure rates are usually in the order of  $10^{-5}$  to  $10^{-6}$ , ultra-dependable applications require the system as a whole to be more reliable than any one of its components. This can only be achieved by utilizing fault-tolerant strategies that enable the continued operation of the system in the presence of component failures.

Secondly, shrinking geometries, lower power voltages, and higher frequencies result in a significant increase of transient failure rates. As a consequence, fault-tolerance mechanisms for improving the reliability of systems in the presence of transient faults also become important for improving the robustness in non safety-critical applications.





## References

- [Aer91] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Specification 651: Design Guide for Integrated Modular Avionics*, November 1991.
- [AUT06] AUTOSAR GbR. *AUTOSAR – Technical Overview V2.0.1*, June 2006.
- [Avi75] A. Avizienis. Fault-tolerance and fault-intolerance: Complementary approaches to reliable computing. In *Proc. of the Int. conference on Reliable software*, pages 458–464, 1975.
- [Bos91] Robert Bosch GmbH, Stuttgart, Germany. *CAN Specification, Version 2.0*, 1991.
- [Buc05] *Avionics Databases*. Avionics Communications Inc., 3rd edition edition, 2005. ISBN 1885544235.
- [C. 02] C. Jones et al. Final version of the DSoS conceptual model. *DSoS Project (IST-1999-11585)*, December 2002.
- [CAN05] CANopen application layer and communication profile v4.0.2. Technical report, CiA DS 301, March 2005.
- [Col99] R.P.G. Collinson. Fly-by-wire flight control. *Computing & Control Engineering Journal*, 10:141–152, August 1999.
- [Cri91] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.
- [Int93] Int. Standardization Organisation, ISO 11898. *Road vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High-Speed Communication*, 1993.
- [KO02] H. Kopetz and R. Obermaisser. Temporal composability. *Computing & Control Engineering Journal*, 13:156–162, August 2002.
- [Kop97] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.
- [Kop99] H. Kopetz. Elementary versus composite interfaces in distributed real-time systems. In *Proc. of the Int. Symposium on Autonomous Decentralized Systems*, Tokyo, Japan, March 1999.
- [Kop01] H. Kopetz. The temporal specification of interfaces in distributed real-time systems. In *Proc. of the EMSOFT 2001*, pages 223–236, Tahoe City, California, USA, October 2001.
- [Kop06] *ARTEMIS Final Report on Reference Designs and Architectures – Constraints and Requirements*. ARTEMIS (Advanced Research & Technology for EMbedded Intelligence and Systems) Strategic Research Agenda, 2006. <http://www.artemis-sra.eu>.
- [KR93] H. Kopetz and J. Reisinger. The non-blocking write protocol NBW: A solution to a real-time synchronisation problem. In *Proc. of the 14th Real-Time Systems Symposium, Raleigh-Durham, North Carolina, USA*, 1993.
- [LA90] P.A. Lee and T. Anderson. *Fault Tolerance Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer Verlag, 1990.

- [LH94] J.H. Lala and R.E. Harper. Architectural principles for safety-critical real-time applications. In *Proc. of the IEEE*, volume 82 of *I*, pages 25–40, January 1994.
- [LS04] J. Leohold and C. Schmidt. Communication requirements of future driver assistance systems in automobiles. In *Proc. of the IEEE Int. Workshop on Factory Communication Systems*, pages 167–174, September 2004.
- [LSPS04] D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk. On the design and development of a customizable embedded operating system. In *Proc. of the International Workshop on Dependable Embedded Systems*, Florianopolis, Brazil, October 2004.
- [OPT07] R. Obermaisser, P. Peti, and F. Tagliabo. An integrated architecture for future car generations. *Real-Time Systems Journal*, 36:101–133, 2007.
- [Pol96] S. Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, 1996.
- [Rep05] Study of worldwide trends and r&d programmes in embedded systems in view of maximising the impact of a technology platform in the area. Technical report, FAST GmbH, Technical University Munich, November 2005.
- [Rus99] J. Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999.
- [Rus01a] J. Rushby. Bus architectures for safety-critical embedded systems. In Tom Henzinger and Christoph Kirsch, editors, *Proc. of the First Workshop on Embedded Software (EMSOFT 2001)*, volume 2211 of *Lecture Notes in Computer Science*, pages 306–323, Lake Tahoe, CA, October 2001. Springer-Verlag.
- [Rus01b] J. Rushby. Modular certification. Technical report, Computer Science Laboratory SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, USA, September 2001.
- [Sif05] J. Sifakis. A framework for component-based construction. In *Proc. of 3rd IEEE Int. Conference on Software Engineering and Formal Methods (SEFM05)*, pages 293–300, September 2005.
- [TAP02] D.A. Thomas, K. Ayers, and M. Pecht. The ‘trouble not identified’ phenomenon in automotive electronics. *Microelectronics Reliability*, 42:641–651, 2002.
- [Vol03] Volkswagen AG. HIS/VectorCAN driver specification 1.0. Technical report, Berliner Ring 2, 38440 Wolfsburg, August 2003.

# AN INTEGRATED ARCHITECTURE FOR FUTURE CAR GENERATIONS

*Real-Time Systems Journal, Volume 36, 2007, pages 101–133, Springer.*

Roman Obermaisser  
Vienna University of Technology  
Real-Time Systems Group  
romano@vmars.tuwien.ac.at

Philipp Peti  
Vienna University of Technology  
Real-Time Systems Group  
peti@vmars.tuwien.ac.at

Fulvio Tagliabo  
Centro Ricerche Fiat  
Orbassano, Italy  
fulvio.tagliabo@crf.it

**Abstract** The DECOS architecture is an integrated architecture that builds upon the validated services of a time-triggered network, which serves as a shared resource for the communication activities of more than one application subsystem. In addition, encapsulated partitions are used to share the computational resources of Electronic Control Units (ECUs) among software modules of multiple application subsystems. This paper investigates the benefits of the DECOS architecture as an electronic infrastructure for future car generations. The shift to an integrated architecture will result in quantifiable cost reductions in the areas of system hardware cost and system development. In the paper we present a current federated Fiat car E/E architecture and discuss a possible mapping to an integrated solution based on the DECOS architecture. The proposed architecture provides a foundation for mixed criticality integration with both safety-critical and non safety-critical subsystems. In particular, this architecture supports applications up to the highest criticality classes ( $10^{-9}$  failures per hour), thereby taking into account the emerging dependability requirements of by-wire functionality in the automotive industry.

**Keywords:** real-time systems, system architectures, automotive electronics, communication networks, legacy systems, dependability, component-based integration

## 1. Introduction

One can distinguish two classes of systems for distributed applications, namely *federated* and *integrated systems*. In a federated system, each application subsystem has its own dedicated computer system, while an integrated system is characterized by the integration of multiple application subsystems within a single distributed computer system. Federated systems have been preferred

for ultra-dependable applications due to the natural separation of application subsystems, which facilitates fault-isolation and complexity management.

Integrated systems, on the other hand, promise massive cost savings through the reduction of resource duplication. In addition, integrated systems permit an optimal interplay of application subsystems, reliability improvements with respect to wiring and connectors, and overcome limitations for spare components and redundancy management. An ideal future system architecture would thus *combine the complexity management advantages of the federated approach, but would also realize the functional integration and hardware benefits of an integrated system* [Ham03, p. 32]. The challenge is to devise an integrated architecture that provides a framework with generic architectural services for integrating multiple application subsystems within a single, distributed computer system, while retaining the error containment and complexity management benefits of federated systems.

The benefits of integrated architectures are becoming more and more important for the automotive domain, since there is a steady increase in electronics in automotive systems in order to meet the customer's expectation of a car's functionality [He04]. Cars are no longer simple means of transportation but rather need to convince customers with respect to design, performance, driving behavior, safety, infotainment, comfort, maintenance, and cost. In particular during the last decade, electronic systems have resulted in tremendous improvements in passive and active safety, fuel efficiency, comfort, and on-board entertainment. In combination with a "1 Function – 1 Electronic Control Unit (ECU)" design philosophy that is characteristic for federated architectures, these new functionalities have led to electronic systems with large numbers of ECUs and a heterogeneity of communication networks.

However, in order to satisfy the industrial demands on performance, dependability and cost with respect to a large variety of different car platforms, the current state-of-the-art system development methodology is heavily imposed to be reviewed, because of

- the strong competition among the carmakers;
- the requirement to continuously improve comfort functionality with stringent time-to-market constraints;
- the introduction of by-wire vehicle control and those functions introduced following normative pressure (e.g., fuel consumptions);
- a demand of greater versatility of the vehicle, conceived in a new view about modularity and standardization.

In particular, a low number of ECUs offers significant benefits with respect to architecture complexity, wiring, mounting, hardware cost and many others. Thus, a reduction of the number of ECUs is of great interest.

It is the objective of this paper to present the DECOS architecture for dependable embedded control systems for future automotive systems. This integrated architecture is based on a time-triggered core architecture and a set of high-level services that support the execution of newly developed and legacy applications across standardized technology-invariant interfaces. Rigorous encapsulation guarantees the independent development, seamless integration, and operation without unintended mutual interference of the different application subsystems. The integrated architecture offers an environment to combine both safety-critical and non safety-critical subsystems within a single distributed computer system. The architecture exploits the encapsulation

services to guarantee that software faults cannot propagate from non safety-critical subsystems into subsystems of higher criticality.

In this paper, we map a present automotive infrastructure onto the DECOS architecture and elaborate on the respective benefits, such as independent development, the assignment of integration responsibility, and an optimized use of resources through architectural gateway services. In order to maximize the economic impact, we propose a portable architecture that can be deployed in all segments of the car manufacturer (segment from A to E and also for luxury cars). Thereby a reduction of cost due to the increase in volume is expected. To achieve the required flexibility and portability, the architecture is based on general purpose hardware and a modular software concept allowing to protect the intellectual property of vendors.

This paper is structured as follows. Section 2 investigates the advantages and disadvantages of federated and integrated architectures. Section 3 gives an overview of the DECOS integrated architecture, presenting the main underlying concepts. In Section 4, we discuss the current state-of-the-art of automotive architectures. The mapping to an integrated solution is the focus of Section 5. The discussion presented in Section 6 evaluates the integrated architecture based on prevalent E/E automotive trends. Section 2 is devoted to an overview of related work on integrated system architectures.

## 2. Federated vs. Integrated Architectures

One can distinguish two extreme classes of architectural paradigms for distributed applications, namely federated and integrated architectures. Real systems are often positioned between these extremes, leaning more to one or the other side. In a totally federated system, each application subsystem (e.g., multimedia domain in a car) has its own dedicated computer system, while an integrated system is characterized by the integration of multiple application subsystems within a single distributed computer system.

### Advantages of Federated Systems

Although the federated system approach has significant deficiencies compared to the integrated systems approach, the federated system approach is superior with respect to complexity control, independent development of application subsystems, intellectual property protection, and exterior error containment.

**Fault Containment.** Based on the assumption that hardware faults effect an entire ECU, which is accepted for ultra-dependable systems [LH94, Kop03], federated systems offer advantages with respect to fault containment. When a hardware fault affects an ECU of a federated computer system, the fault impacts only a single application subsystem. Conversely, a hardware fault affecting an ECU of an integrated system can impact multiple application subsystems, because an ECU in an integrated system can be shared among software modules of multiple application subsystems. For developmental faults, better fault containment of a federated system in comparison with an integrated system is a consequence of the platform heterogeneity. The diversity of the employed hardware (e.g., processors, boards, etc.) and software platforms (e.g., operating systems) of the different application subsystems limits the regions of the immediate impact of developmental hardware and software faults.

**Error Containment.** Exterior error containment addresses error propagation between application subsystems. Since a federated system implements application subsystems via separate

computer systems, a computer system for an application subsystem remains functional despite the failure of other computer systems and the corresponding application subsystems. However, error containment within the application subsystem is no mere consequence of the federated system approach, but must be provided by the architecture or the application.

**Independent Development.** The ability for independent development follows from the near independence of federated systems. In federated systems, only a very limited level of interactions occurs via gateways, thus keeping the need for coordination between different vendors down to a minimum.

**Complexity Control.** The implementation of each application subsystem on a dedicated distributed computer system with controlled interactions across gateways also helps in managing complexity. The hiding of the internals of application subsystems enables a designer to understand the behavior of any particular application subsystem in isolation, i.e., without analyzing and fully understanding the rest of the system.

### Advantages of Integrated Systems

Among the major advantages of integrated systems are hardware cost reduction and dependability improvements.

**Hardware Cost Reduction.** In contrast to federated systems, which require a dedicated computer system for each application subsystem, integrated systems facilitate the multiplexing of hardware resources. In the automotive area, the trend of federated architectures with increasing numbers of ECUs is coming to its limits, because systems are becoming too complex and too costly with the current practice of having each ECU dedicated to a single function.

**Dependability Improvements due to Reductions of Wiring and Connectors.** By tackling the “1 Function – 1 ECU” problem, the reduction in the overall number of ECUs also leads to increased reliability by minimizing the number of connectors and wires. Field data from automotive environments has shown that more than 30% of electrical failures are attributed to connector problems [SM99].

**Fault-Tolerance.** Replicated hardware is necessary to tolerate hardware faults in both federated and integrated systems. However, in a federated system these resources are available only to a single computer system out of the numerous loosely coupled ones. A computer system dedicated to a particular application subsystem can fail, although the overall number of spare ECUs would be sufficient to tolerate a given number of ECU failures. In an integrated system, resources are universally available among the different application subsystems.

## 3. The DECOS Integrated Architecture

The DECOS architecture [OPHS06] offers a framework for the design of large embedded real-time systems by physically integrating multiple application subsystems on a single distributed computer systems. The DECOS architecture distinguishes clearly between logical and physical structuring (top and bottom levels in Figure 1). Structuring rules guide the designer in the decomposition of the overall system at the logical level and support the transformation to the physical

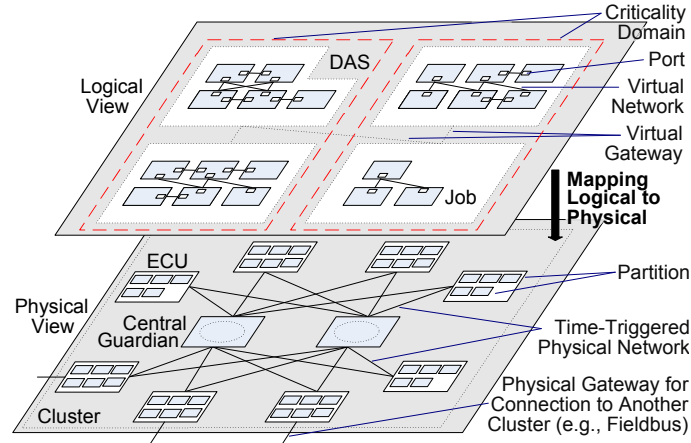


Figure 1. Structuring of a DECOS System – Logical and Physical View

level. Furthermore, the DECOS architecture offers to system designers generic architectural services, which provide a validated stable baseline for the development of applications.

### Logical View of a DECOS System

A DECOS system (e.g., the complete on-board electronic system of a car) provides to its users (e.g., human operator) application services at the controlled object interface. By means of modularization, the DECOS system is decomposed into a set of nearly-independent *DASs*. Each DAS provides a subset of the overall application services, which is meaningful in the application context. An example for a DAS in the automotive domain would be a steer-by-wire subsystem [Hei03].

We further decompose each DAS into smaller units called *jobs*. A *job* is the basic unit of work and employs a *virtual network* in order to exchange messages with other jobs and work towards a common goal. The access point between a job and a virtual network is called a *port*.

### Physical View of a DECOS System

From a physical point of view (see bottom level in Figure 1), a DECOS system encompasses a cluster containing a set of *integrated node computers* (also called ECUs), which are interconnected by a time-triggered physical network. The virtual networks introduced in the logical view are implemented on top of this time-triggered physical network. The use of a time-triggered physical network matches the predictability and fault-tolerance requirements of safety-critical applications [Rus01].

Every ECU provides one or more *partitions*, each hosting a corresponding job. A partition is an encapsulated execution space within an ECU with a priori assigned computational (e.g., CPU, memory, I/O) and communication resources (e.g., network bandwidth, latencies). By supporting the deployment of multiple jobs on one ECU, the DECOS architecture goes beyond the prevalent “1 Function – 1 ECU” design methodology [BS05, GFL<sup>+</sup>02].

In addition, a DECOS system can contain connections to external systems, such as a fieldbus network or a legacy cluster. For this purpose, ECUs implement so-called *physical gateways*.

## Architectural Services

The DECOS architecture offers generic architectural services that separate the application functionality from the underlying platform technology. The architectural services facilitate reuse and reduce design complexity. The specification of the architectural services hides the details of the underlying platform, while providing all information required for ensuring the functional and meta-functional (dependability, timeliness) requirements in the design of a safety-critical real-time application. Applications build on an architectural service interface that can be established on top of numerous platform technologies.

In order to maximize the number of platforms and applications that can be covered, the DECOS architecture distinguishes a set of three *core services* and an open-ended number of *high-level services* that build on top of the core services. The core services are as follows:

- 1 **Deterministic and Timely Transport of Messages.** This service performs periodic time-triggered exchanges of state messages. TDMA controls the media access to the replicated communication channels and a communication schedule determines the global points in time of all message transmissions. Slots are statically assigned to ECUs in a way that allows each of them to send a message during a communication round.
- 2 **Fault-Tolerant Clock Synchronization.** In a distributed computer system, ECUs capture the progression of time with physical clocks containing a counter and an oscillation mechanism. An observer can record the current granule of the clock to establish the *timestamp* of an event. Since any two physical clocks will employ slightly different oscillators, the time-references generated by two clocks will drift apart. Clock synchronization is concerned with bringing the time of clocks in a distributed system into close relation with respect to each other.
- 3 **Strong Fault Isolation.** Although a Fault Containment Region (FCR) can demarcate the immediate impact of a fault, fault effects manifested as erroneous data must be prevented from propagating across FCR boundaries [LH94]. For this reason, the DECOS architecture provides error containment for failures recognized within the fault hypothesis [OP06]. In addition, DECOS offers a reliable distributed computing platform by ensuring that a message is either consistently received by all correct ECUs or detected as being faulty at all correct ECUs.

Any architecture with a time-triggered physical network that provides these core services (e.g., Time-Triggered Architecture (TTA) [KB03], FlexRay [Fle05], Time-Triggered Ethernet (TTE) [KAGS05]) can be used as a basis for the implementation of the DECOS integrated architecture. The small number of core services eases a thorough validation, which is crucial for preventing common mode failures as all high-level services and consequently all applications build on the core services.

Based on the core services, the DECOS integrated architecture realizes high-level architectural services, which are DAS-specific and constitute the interface for the jobs to the underlying platform. Among the high-level services are the virtual network services [OP05], the gateway services [OP05b], and the diagnostic services [PO06]. For example, the virtual network services establish on top of the core services the communication infrastructure of a DAS as an overlay network (i.e., denoted as a virtual network). On top of the time-triggered physical network, multiple virtual networks with different protocols can be established, such as a virtual network executing



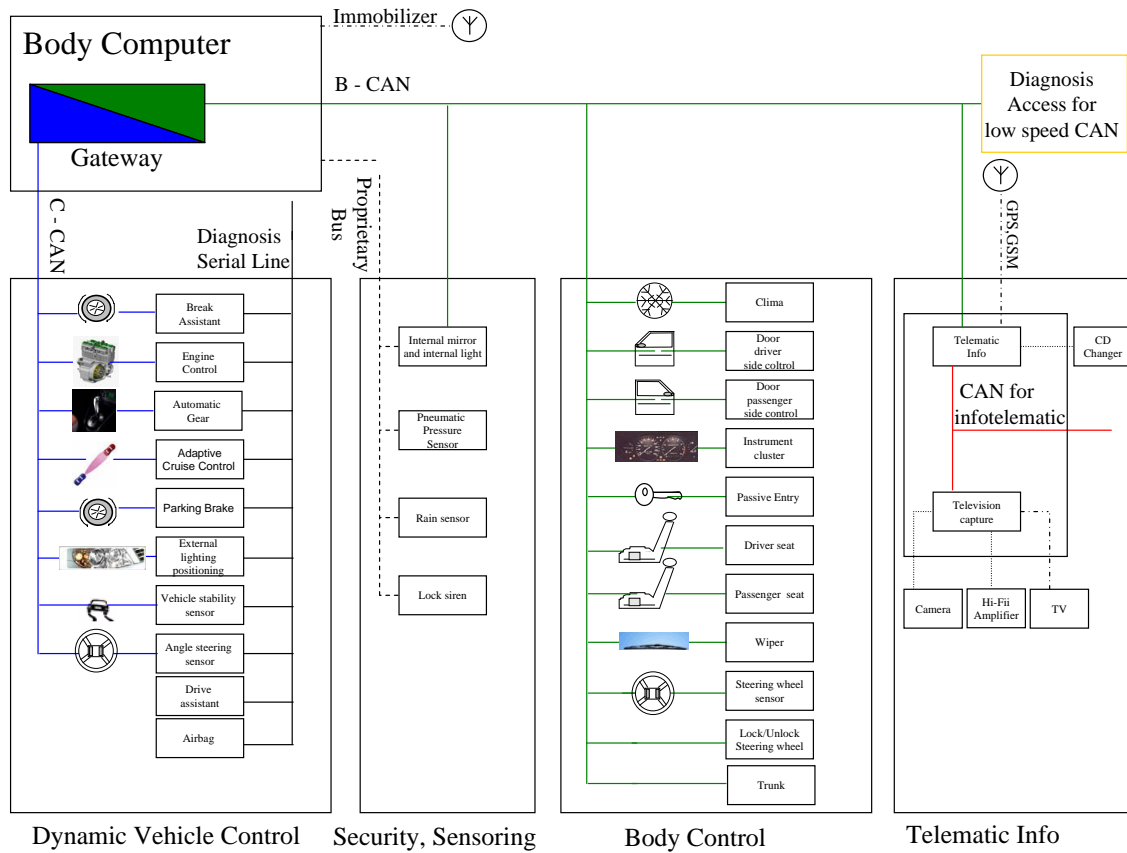


Figure 2. The Electronic Infrastructure of a Fiat Car

the CAN protocol for a non safety-critical DAS (e.g., comfort) or a time-triggered virtual network for a safety-critical DAS (e.g., X-by-wire).

#### 4. Today's Automotive Architectures

To give an impression of the complexity and the amount of electronics in today's luxury cars take for example the electronic infrastructure of a Fiat car depicted in Figure 2. The distributed ECUs of each federated cluster of the car are interconnected via communication networks with different protocols (e.g., Controller Area Network (CAN) [Bos91], Local Interconnect Network (LIN) [LIN03]), physical layers, bandwidths (10 kbps–500 kbps), and dependability requirements.

The *Body Control* cluster as well as the *Telematic Info* cluster, are typically implemented via a low speed CAN bus (125 kbps), while the *Dynamic Vehicle Control* cluster is implemented via a high speed CAN bus (500 kbps). The *Infotelematic* system also uses a high speed CAN to exchange camera and video information. These multiple federated clusters are interconnected with a central gateway inside the *Body Computer*, allowing controlled data exchange between the *Dynamic Vehicle Control* cluster and the *Body Control* cluster, and access to the On-Board Diagnosis (OBD) system of each ECU. For on-board diagnosis either a dedicated serial line interconnects the ECUs or the diagnostic protocol is executed via the low speed CAN network.

Each of these clusters consists of ECUs that are typically dedicated to a single job. In combination with the need to adapt products to emerging trends and customer requests, manufactures are forced to steadily increase the number of deployed ECUs in order to improve the functionality of the car. For example, Fiat cars contain up to 40 ECUs. However, this trend of increasing the number of ECUs is coming to its limits, because of complexity, wiring, space and weight restrictions. For example, electrical connections are considered to be one of the most important failure causes in automotive systems. Field data from automotive environments has shown that more than 30% of electrical failures are attributed to connector problems [SM99]. With an average cost of 30-50 Euros per ECU this high number of ECUs also bears significant potential for cost reduction.

## Development Process

The typical development process in the automotive industry follows the V-cycle model [Rei06, GFL<sup>+</sup>02]. The OEM acts as the overall system architect by defining the electronic architecture of the car. This includes the definition of the physical structure (e.g., ECUs, wiring, connectors, packaging) and the provided services (e.g., data dictionary for exchanged messages).

Based on this specification, the sourced suppliers (or in-house development teams) implement the respective ECUs. In general, the development of each ECU involves tight interactions between the suppliers and the OEM. Right from the beginning of the development process, the OEM monitors the progress of the suppliers in order to ensure correctness and timely delivery of the ECUs.

For the validation, both component and system integration tests are performed at the OEM. During the component test, each ECU is validated in isolation. For example, a CAN-based ECU is provided with messages from a rest bus simulator. For simulating the I/O, typically Hardware-in-the-Loop (HiL) is used. During system integration test, the interplay of multiple ECUs is tested (e.g., body domain HiL simulator). One focus of the system integration test is the evaluation of the emerging services of the ECUs, i.e., those services that are provided by more than one ECU (e.g., advanced parking assistant). Another focus is the validation whether the prior services of the ECUs are still correct after integration. For example, the temporal specification must not be violated (e.g., timeouts).

However, today during system integration significant efforts are caused by unanticipated interactions between subsystems provided by different vendors. The sharing of communication resources in today's cars across different subsystems (e.g., systems based on the CAN protocol) makes it hard to fully test the functionality of a subsystem in isolation as it will be integrated in the car. As a consequence, there is the need for a comprehensive integration test by the car manufacturer to determine possible mutual interference of subsystems. In contrast, a system architecture with rigorous operational interface specification [KS03] and error containment can avoid the introduction of mutual interference during system integration. Such a temporally composable architecture [KO02] exhibits the benefit of dramatically decreasing integration costs, because the validity of test certificates from suppliers is not invalidated during system integration.

## Complexity Control

Each subsystem (e.g., engine control, brake assistant) possesses a functional complexity that is inherent to the application. The functional complexity of a subsystem when implemented on a target system is dramatically increased in case the architecture does not prevent unintended architecture-induced side effects at the communication system. Since federated systems employ a dedicated computer system for each subsystem, the complexity of the system is lower compared to the integrated systems approach. The absence of interactions and dependencies between sub-

systems reduces the cognitive complexity to a manageable level. In today's cars we do not find a totally federated architecture nor an integrated one. In fact, the economic pressure in the automotive industry requires system designers to utilize the available communication resources for more than one subsystem without protecting the resources from mutual interference. For a deeper understanding consider an exemplary scenario with two subsystems. If the two subsystems share a common CAN bus [Bos91], then both subsystems must be analyzed and understood in order to reason about the correct behavior of any of the two subsystems. Since the message transmissions of one subsystem can delay message transmission of the other DAS, arguments concerning the correct temporal behavior must be based on an analysis of both subsystems. In a totally federated system, on the other hand, unintended side effects are ruled out, because the two subsystems are assigned to separate computer system.

## 5. An Architecture for Future Car Generations

This section describes the proposed integrated architecture for future car generations. We start by discussing a hybrid top-down/bottom-up design strategy, improving the currently prevalent ECU-centric development process. The decomposing during the process results in a set of DASs. Thereafter, we elaborate on the DASs of a hypothetical future car and its constituting gateways. Finally, we show the physical structure of the integrated system.

### Design Flow

The design flow of automotive distributed systems can be decomposed into three phases, the requirement analysis, the subsystem design, and the system integration phase [GFL<sup>+</sup>02] (see also Figure 3). As described in [RH04, SW04] an ECU-centric design process prevails in the automotive industry. Such a bottom up process, however, bears significant drawbacks such as resource duplications, local instead of global quality-of-service optimization, and exponential growth in terms of system integration costs. Furthermore, the number of the deployed ECUs steadily increases to satisfy recent market trends and the customer's demand for new functionality.

The DECOS architecture, by contrast, also supports a top-down design approach. During the requirement analysis the system integrator captures the requirements of the overall system (i.e., the car electronics) and decomposes the system into nearly-independent subsystems (i.e., DASs). The requirement analysis provides the foundation for all later design stages. Here, the overall functionality of the system is specified and subsystems are identified to enable an independent development of DASs. As depicted in Figure 3 the result of this design phase is a set of DASs that comprise the electronic infrastructure of the car.

The structuring of the overall application functionality into DASs is guided by the following principles:

- 1 **Functional Coherence.** A DAS should provide a meaningful application service (e.g., brake-by-wire service of a car) to its users at the controlled object interface. By associating with a DAS an application service that is relevant in the actual application context, the mental effort in understanding the various application services is reduced. An application service can be analyzed by solely considering the jobs of the DAS, the interactions to the controlled object and the gateways to other DASs (inter-DAS interfaces). In particular, it is not necessary to possess knowledge about the internal behavior of DASs, other than the one providing the application service that is of interest.

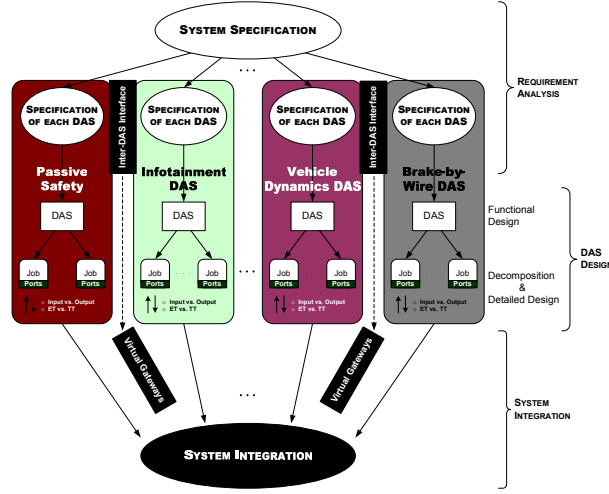


Figure 3. Design Flow

2 **Common Criticality.** In general, the realization of safety-critical services is fundamentally different from the design of non safety-critical services. While the first incorporate fault-tolerance functionality and focus on maximum simplicity to facilitate validation and certification, the latter are usually characterized by a larger amount of functionality and the requirement of flexibility and resource efficiency. The integrated architecture takes this difference into account by distinguishing between safety-critical and non safety-critical DASs along with dedicated architectural services.

3 **Infrastructure Requirements.** A DAS possesses common requirements for the underlying infrastructure. A single virtual network is employed for exchanging message within the DAS. Consequently, common requirements (e.g., with respect to dependability, bandwidth and latency requirements, flexibility) are a prerequisite for deciding on a particular virtual network protocol (e.g., time-triggered or event-triggered) and a corresponding configuration (e.g., bandwidth).

Whenever significant differences in the above aspects are present, such as missing functional coherence or differences with respect to the infrastructure requirements, a DAS is split into smaller DASs for resolving these mismatches.

This *divide and conquer* principle can only be realized if the dependencies between DASs are made explicit in order to avoid hidden interactions (e.g., via the controlled object) that may prevent a seamless system integration. These DASs are then assigned and independently developed by different vendors. In general, each vendor may also depend on subcontractors to deliver the subsystem.

In order to ensure correct system integration the specification of inter-DAS relationships is of high importance. Inter-DAS interfaces as indicated in Figure 3 are used to specify common information within DASs (e.g., sensor information), possible interrelationships via the controlled object, and meta-functional aspects. This way, resources can be shared among DASs, thus avoiding resource duplication by eliminating sensors or using redundant sensory information to improve dependability.

The DAS design is typically performed by different vendors with expert knowledge in particular application domains (e.g., infotainment, braking systems). Independent development of a DAS allows to adopt the benefits of the federated systems design approach to be incorporated into the integrated systems design approach.

Finally, the system integrator needs to unify the separately developed subsystems into the overall system. System integration unites the separately developed subsystems into the overall system. An integrated system approach must provide solutions that reduce integration time and efforts (and consequently reduce integration costs). Smooth system integration is only possible, if the inter-DAS interfaces have been precisely specified and all vendors have performed implementations adhering to these interface specifications. During system integrations three main tasks need to be performed by the system integrator: the physical allocation of the jobs (of all DASs) to partitions taking dependability and resource constraints into account, the configuration of the virtual communication networks, and the realization of the virtual and physical gateways in order to provide emerging services.

## Integrated System Structure of Car

In this subsection, we map the introduced electronic infrastructure of today's cars onto the DECOS integrated architecture. In addition, we replace state-of-the art powertrain domain functionality with by-wire subsystems to emphasize the suitability of the proposed DECOS architecture for mixed criticality applications (i.e., safety-critical and non safety-critical applications). We split up the existing domains (e.g., powertrain, body) into smaller DASs. Smaller DASs are a key element to achieve the DECOS goals with respect to complexity management, independent development and error containment.

As described in Section 3, a DAS represents a nearly-independent subsystem [Sim96, chap. 8], because it can be understood independently from the detailed structure of other DASs, i.e., only based on the specification of the jobs of the DAS and the gateways to other DASs. The controlled export of information through gateways enables the designer to abstract from the jobs in other DASs, considering only the interface specification of the gateways [OP05b].

The DECOS architecture encapsulates DASs both at the level of the communication activities (through encapsulated virtual network services [OP05]) and at the level of the computational activities (partitions in application computers [OPHS06]). Therefore, a design fault in a DAS (e.g., a job with a babbling idiot failure [Kop97]) cannot affect the communication resources (e.g., bandwidth, guarantee of latencies) and computational resources (e.g., CPU time) available to other DASs. Naturally, the finer the subdivision into DASs, the more effective the encapsulation through the architecture becomes.

In a federated architecture, which assigns each DAS to its own dedicated computer system, a strategy with a large number of small DASs would not be feasible due to the cost resulting from increasing resource duplication (via separate networks and ECUs). However, in our proposed integrated architecture, the resulting larger number of DASs does not induce a larger number of physical networks and ECUs. Each DAS is provided as a virtual network on top of the physical time-triggered network of the DECOS architecture. Similarly, the virtual gateways (see Section 5) for coupling DASs do not induce any additional ECUs and connectors.

Based on this line of reasoning, we introduce a finer granularity of DASs compared to the domains of today's automotive architectures (see Figure 4):

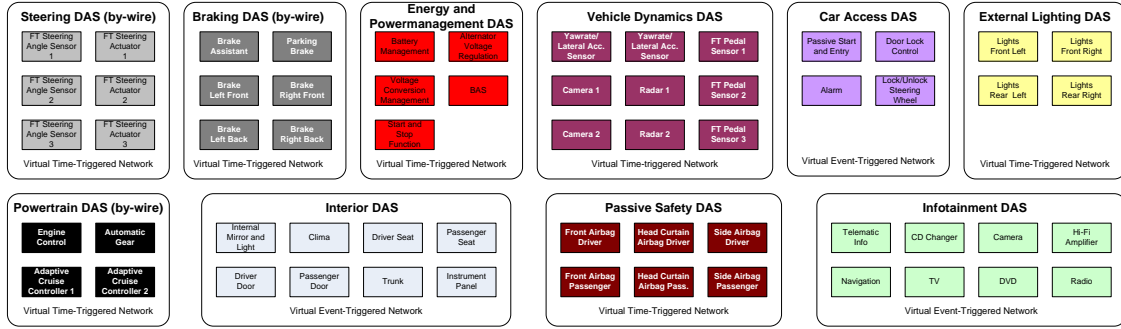


Figure 4. The Distributed Application Subsystems of the Integrated Automotive System

**Steering DAS.** With steer-by-wire the transmission of the wheel rotation to a steering movement of the front wheel is performed with the help of electronically controlled actuators at the front axle [Hei03]. The main advantages in comparison with conventional steering systems are improvements with respect to crashworthiness, weight, and interior design.

**Powertrain DAS.** The functionality of this DAS includes engine control, automatic gear control, and adaptive cruise control.

**Braking DAS.** The braking DAS comprises the brake-by-wire functionality [HB98]. Brake-by-wire systems remedy deficiencies of conventional hydraulic braking systems, such as aging of braking fluids, difficulties in routing of pipes, and the inconvenient feedback during ABS braking. Brake-by-wire systems incorporate brake power assist, vehicle stability enhancement control, parking brake control, and tunable pedal feel.

**Vehicle Dynamics DAS.** In the vehicle dynamics DAS all sensory information that is relevant for controlling the dynamics of the vehicle is captured. By exporting these real-time images to the other DASs of the system the problem of resource duplication can be significantly reduced. In addition, sensor-fusion algorithms [Elm02] can combine the measurements of different sensors to obtain more accurate real-time images.

**Energy DAS.** The main purpose of this DAS is the optimization of the power distribution (electrical energy and power management techniques) for conserving the power available in the vehicle.

**Passive Safety DAS.** The passive safety DAS intends to keep the passengers in the car and effectively decelerates the occupants in order to minimize harm in case of a crash.

**Car Access DAS.** The functionality of this DAS includes a passive keyless access and start system with theft alert. The driver carries an identification device to control the door lock and can start the vehicle by pressing a button.

**Interior DAS.** This DAS comprises the body electronics of the passenger compartment and accesses fieldbus network, such as those embedded in the doors and seats of the car. The functionality of this DAS includes the control of the doors (e.g., mirrors, window lifters), the seats

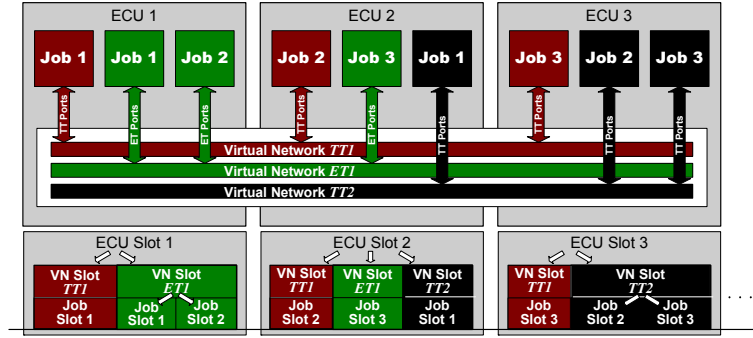


Figure 5. Hierarchic Subdivision of TDMA Slots

(e.g., seat adjustment, the position memory), the instrument panel, the climate control, and the lighting of the passenger compartment.

**Infotainment DAS.** Car drivers are no longer satisfied with cars being simple means of transportation. Today's luxury cars include GPS navigation systems, DVD players and high-end audio systems. In addition, voice control and hands-free speaker phones relieve the driver from concentrating on multimedia devices instead of traffic.

**External Lighting DAS.** This DAS encompasses jobs for controlling the rear lights (e.g., rear fog lamps, braking lights) and front lights (e.g., indicators, dipped beam, main beam, front fog lamps). In addition, the jobs of the external lighting DAS control the position of the adaptive forward lighting.

## Virtual Networks

Each DAS is provided with a dedicated communication infrastructure that is realized as a *virtual network*. A virtual network is established as an overlay network on top of a time-triggered physical network [OP05].

The establishment of a virtual network occurs through a hierarchic temporal subdivision of the communication resources provided by the time-triggered communication protocol. The media access control strategy of the time-triggered communication protocol is TDMA. TDMA statically divides the channel capacity into a number of slots and controls access to the network solely by the progression of time. An ECU broadcasts messages during its ECU slot and receives messages during the slots of the other ECUs. A virtual network further subdivides the ECU slots by assigning to each job that is located on an ECU a respective job slot (see Figure 5). For transmitting the messages produced by a job, the virtual network service uses this job slot in the TDMA scheme.

Virtual networks follow rigorously a TDMA schedule, in which each job sends during an interval of time with a priori known start and end instants w.r.t. to a global time base. This static communication schedule has the benefit of enabling error containment with respect to the communication resources. The architecture can ensure that each job writes only into its own communication slots, e.g., by using guardian functionality with a fault-tolerant time-triggered network in conjunction with middleware as discussed in [OP05]. Thereby, a faulty job is prevented from affecting the data integrity and temporal properties (e.g., latency) of the messages sent by other jobs.

Using job slots in a TDMA scheme, virtual networks with higher protocols have been established on top of a time-triggered physical network. Examples are time-triggered virtual networks [OP05], virtual CAN networks [OP05a], a transport protocol for a hard-real time CORBA broker [SLO03], and virtual networks with TCP/IP [Nex03].

The selected protocol depends on the respective criticality and regularity of the communication activities. Time-triggered virtual networks are well-suited to handle the communication exchanges of all safety-critical and safety-relevant DASs. It has become widely accepted that safety-critical automotive applications employ the time-triggered control paradigm [Rus01], because this control paradigm permits to guarantee a deterministic behavior of all safety-related message transmissions even at peak-load. In addition to hard real-time performance time-triggered control also supports temporal composability and facilitates the realization of fault-tolerance mechanisms. For this reason the communication infrastructure for the steer-by-wire, the brake-by-wire, the powertrain, the vehicle dynamics, the passive safety and the external lighting DAS are *time-triggered virtual networks*.

Event-triggered virtual networks like CAN, on the other hand, are the communication infrastructure of choice for those DASs having less stringent dependability requirements. Here, the flexibility and the efficient use of resources is more important than the determinism provided by the time-triggered control paradigm. For this reason, the jobs of the interior, power management, car access, and infotainment DAS are interconnected by respective *event-triggered virtual networks* in the presented architecture.

## Gateways

By splitting the overall functionality of the car into multiple DASs the need for a coupling of individual DASs emerges. The presented architecture supports gateways as a generic architectural services for the interconnection of DASs. Gateways have significant advantages with respect to the elimination of resource duplication and the tactic coordination of application subsystems. In a large automotive system, different application subsystems typically depend on the same or similar sensory inputs and computations. Gateways allow to exploit system-wide redundancy of sensor information in order to increase reliability or reduce resource duplication. In addition, gateways permit the coordination of DASs in order to improve quality of control.

In the DECOS integrated architecture, we sharply distinguish between architecture level and application level. Based on this differentiation, we can identify two choices for the construction of a gateway. A hidden gateway performs the interconnection of virtual networks at the architecture level. Generic architectural services – although parameterized by the application requirements – are transparent to the jobs at the application level. A visible gateway, on the other hand, performs the interconnection at the application level.

A *virtual gateway* [OP05] interconnects two virtual networks of two respective DASs by forwarding information contained in the messages received at the input ports of one virtual network onto the output ports towards the other virtual network.

In general, the semantic and operational properties of the input ports at one virtual network can be different to the semantic and operational properties of the output ports at the other virtual network. The resulting property mismatch [C. 02] is resolved by the gateway by performing transformations on the information passing through the gateway. For syntactic transformations, the gateway requires a description of the syntactic format (i.e., the data types) of the messages passing through the gateway and rules for transforming the different syntactic transformations into each other. If the DASs interconnected by the gateway exhibit different operational speci-



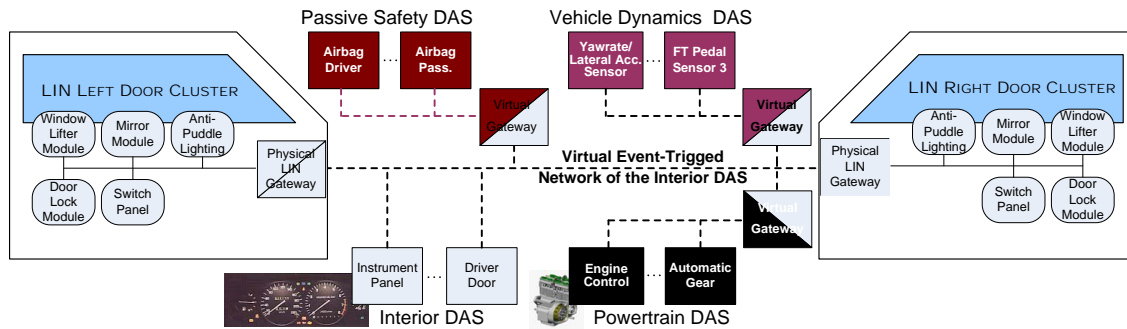


Figure 6. Physical and Virtual Gateways of the Interior DAS

fication styles [KS03], the gateway requires additional buffering functionality for the exchange of information between virtual networks with varying rigidity of temporal specifications. For example, such a scenario occurs, if one DAS operates time-triggered and the second DAS operates event-triggered.

In addition, virtual gateways ensure encapsulation by using a filtering specification in the value and temporal domain [OP05b] in order to restrict the redirection of messages through the gateway. In general, only a fraction of the information exchanged at one virtual network will be required by jobs connected to the virtual network at the other side of the gateway. By restricting the redirection through the gateway to the information actually required by the jobs of the other DAS, the gateway not only improves resource efficiency by saving bandwidth of unnecessary messages, but also facilitates complexity control.

In addition to the interconnection of virtual networks, the presented integrated architecture offers architectural gateway services for interacting with the environment via *physical gateways*. In general, the interaction of the computer system with the controlled object and the human operator can occur either via a direct connection to sensors and actuators or via a fieldbus network. The latter approach simplifies the installation – both from a logical and a physical point of view – at the expense of increased latency of sensory information and actuator control values.

Since the prevalent low-cost fieldbus protocol in the automotive domain is LIN [LIN03], the proposed architecture supports physical LIN gateways, each acting as a master for the slaves of the physical LIN bus. Figure 6, which exemplifies the role of hidden gateways in the logical structure of the future automotive architecture, depicts two of these LIN gateways for the interconnection of the interior DAS with the LIN fieldbusses located at the front doors. The driver door job and passenger door job exchange information with the actuators and sensors in the doors in order to control door locks, window lifters, mirrors, and anti-puddle lighting.

In addition, Figure 6 also contains virtual gateways for the interconnection of virtual networks. The interior DAS constructs real-time images capturing the state of the passenger compartment of the vehicle (e.g., status of the doors, seats, lighting, climate control) that are also important to other DASs. For example, the driver's weight as measured at a seat is an important parameter for the passive safety DAS in order to adapt air bags to different passengers (e.g., children). The current temperature inside and outside the car, which is captured by the climate control subsystem, is another example for a real-time entity that is significant beyond the interior DAS. Temperature measurements are an essential input for physical models of sensors in other DASs (e.g., powertrain DAS) and permit to improve the precision and plausibility of sensory information.

Adversely, other DASs need to be able to control body electronics in the interior DAS. In hazardous situations, e.g., after the detection of a potential crash as indicated by yaw rate and lateral acceleration sensors (e.g., during skidding and emergency braking), the vehicle dynamics DAS causes the tensioning of seat-belts and realigning of seats to a safer positions.

In the following, we will discuss one of the virtual gateways in more detail, namely the gateway for the forwarding of engine information from the powertrain DAS to the interior DAS. Thereby, engine status information (such as engine speed, vehicle speed, fuel indication) is displayed on the instrument panel.

The *engine control job* is part of the powertrain domain. It accesses numerous actuators and sensors to the motor (e.g., cam shaft and crank shaft position). The engine control job receives messages from the vehicle dynamics DAS (e.g., message with engine torque request from ESP) and from the powertrain DAS (e.g., transmission control job). The output of the engine control job are messages with status information of the engine, such as:

*msg E1* : < engine speed (16bit, unit: RPM), throttle position (8bit, unit: percent) >  
*msg E2* : < fuel consumption (16bit, unit: liters/hour), engine oil temp. (8bit, unit: A°C) >

The *instrument panel job* receives messages from the virtual CAN network of the interior DAS and displays the status information contained in these messages on the instrument panel. Among the received messages are:

*msg IP1*: < engine speed (16bit, unit: RPM), engine oil temperature (8bit, unit: °C) >  
*msg IP2*: < vehicle speed (16bit, unit: km/hour), odometer (16bit, unit: km) >  
*msg IP3*: < door status (1bit per door, unit: open/closed) >

The virtual gateway between the powertrain DAS and the interior DAS needs to redirect information from the engine control job to the instrument panel job to be shown to the driver. Also, the gateway needs to resolve a property mismatch between the different communication paradigms of the DASs. In detail, the actions of the gateway are as follows:

**Selective redirection.** The virtual gateway redirects only those signals from the messages with the engine status that are actually required in the interior DAS. For example, the oil temperature and the engine speed in messages *E1* and *E2* are redirected via message *IP1*, while the throttle position in message *E1* is discarded by the gateway.

**Conversion of communication paradigms.** The powertrain DAS employs a time-triggered virtual network, i.e., all messages are exchanged periodically at a priori specified points in time. The interior DAS, on the other hand, possesses an event-triggered virtual network with on-demand message transmissions (i.e., triggered by significant events). The gateway performs time-triggered receptions of engine status messages. The transmission of messages, however, occurs event-triggered. A message is transmitted on the interior DAS only in case the value of the real-time entity has changed. No message is sent in case the real-time entities (e.g., oil temperature) remain unchanged.

**Traffic shaping.** The message transmissions of the engine status information in the powertrain DAS occur at a higher frequency than can be used for updating the instrument panel in the interior DAS. For this reason, the virtual gateway enforces a minimum message interarrival time for the message transmissions on the event-triggered virtual network of the interior DAS.

More details on the formal specification of these actions using timed gateway automata and their implementation can be found in [OP05b].

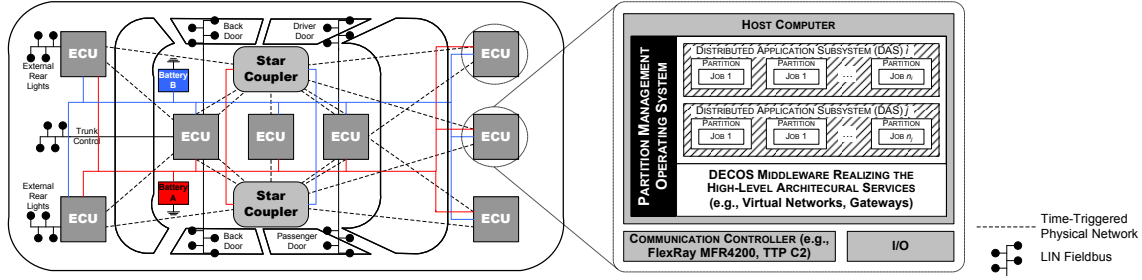


Figure 7. Physical System Structure

## Physical System Structure

The mapping from the logical to the physical system structure needs to assign jobs to ECUs and virtual networks to the time-triggered physical network. In order to tolerate arbitrary single component failures it is mandatory to devise a mapping of jobs to ECUs in a way, that redundant jobs are not hosted on the same ECU. Similarly to the sharing of ECUs among jobs, the time-triggered physical network is the basis for multiple virtual networks. In order to achieve the dependability requirements of safety-critical DASs, the time-triggered physical network relies on two central guardians [BKS03] for achieving fault isolation even in case of arbitrary ECU failure modes. Fault injection experiments have shown that for ultra-dependable applications restrictions concerning the failure modes of ECUs are unjustified [ASBT03].

In order to support the collocation of multiple jobs and the establishment of virtual networks, an ECU incorporates the following structural elements as depicted in Figure 7:

**Partition management operation system.** The host computer of an ECU runs multiple jobs of different DASs. For this purpose, the host computer uses a *partition management operating system*, which establishes for each job a corresponding partition with guaranteed computational resources (CPU time, memory). The partition management operating system implements mechanisms for spatial and temporal partitioning in order to protect the computational resources of the individual partitions. The scheduling of partitions needs to ensure that a timing failure of a job, such as a worst-case execution time violation, does not affect the CPU time available to other partitions. The spatial partitioning mechanisms of the partition management operating system include memory protection between partitions (e.g., hardware-enforced with a Memory Management Unit (MMU)). Thereby, each partition emulates a “virtual ECU” that is dedicated to a single job only. An example for a partition management operating system is a time-triggered operating system based on Linux Real-Time Application Interface (RTAI) [HPOS05]. Further examples of suitable operating systems, which fit into the proposed architecture, are LynxOS-178 [Lyn06] and VxWorks [KWR04]. These two operating systems are certified operating systems that support time partitioning through a fixed-cyclic time-slice scheduler.

**DECOS middleware.** The host computer contains middleware for realizing the high-level architectural services. The purposes of the middleware include the management of the communication resources by means of virtual networks and gateways as previously described. The middleware provides a technology invariant interface to the jobs that abstracts from any hardware-specific implementation details.

**Communication controller.** The *communication controller* executes a time-triggered communication protocol (e.g., FlexRay [Fle05], Time-Triggered Protocol (TTP) [TTT02a]) to offer the deterministic and timely transport of messages, fault-tolerant clock synchronization, and strong fault isolation as described in Section 1.

**I/O.** The jobs hosted on an ECU exploit the input/output subsystem for interacting with the controlled object and the human operator. This interaction occurs either via a direct connection to sensors and actuators or via a fieldbus (e.g., LIN [LIN03]). The latter approach, which simplifies the installation from a logical and physical point of view at the expense of increased latency, is depicted in Figure 7. Physical LIN gateways connect physical LIN clusters to the integrated system, e.g., the LIN fieldbus within the doors of the cars are connected to the ECUs located in the center of the car.

## 6. Benefits of the Proposed Architecture

This section summarizes the main benefits of the proposed integrated automotive architecture and highlights the major design decisions. We will show that the proposed architecture is in line with prevalent architectural trends, such as reuse of functionality across different car segments and introduction of safety-critical applications.

### Economic Benefits

Present day automotive systems follow a federated philosophy with different types of automotive networks [LH02]. The multitude of communication protocols is a result of the different requirements with respect to functionality, dependability, and performance of the automotive DASs, such as the infotainment DAS, the comfort DAS, or the powertrain DAS.

In contrast to these federated architectures, the integrated architecture for future automotive systems presented in this paper allows to share communication and computational resources among different DASs in order to evolve beyond a “1 Function – 1 ECU” strategy and achieve a significant reduction in the overall number of ECUs. With an average cost of 40 Euro per ECU in today’s cars and 0.2 Euro per wire, the total hardware cost of a federated automotive system with 40 ECUs and 800 wires is about 1800 Euro. If we assume that the number of ECUs will be reduced to 30 ECUs and the number of wires to 500 in the integrated system, with an increased average cost per ECU of 50 Euro, then total hardware cost is reduced by 200 Euro per car.

### Complexity

A minimal complexity, i.e., a minimal mental effort to analyze and understand a system, is one of the most important design drivers of the presented integrated architecture. Today, the unmanaged complexity of systems is the major cause of design faults. *Complexity increases the likelihood of serious, yet latent, design flaws* [JB92, p. 39]. In [Lev86, p. 131] it is stated that *complexity of software and hardware causes a non linear increase in human-error induced design faults*. High system complexity also prolongs development, which is detrimental to a company’s economic success, because today’s business realities demand a short time-to-market. In addition, complexity complicates validation and certification as state-of-the-art formal verification tools are limited in the size of a design that they can handle.

In order to manage complexity, the presented integrated architecture enables designers to proceed in such a way as if they were realizing DASs in a federated system. A DAS along with

the corresponding communication resources (virtual networks) and computational resources (partitions in ECUs) is encapsulated and interactions between DASs are limited to the exchange of messages via precisely specified hidden gateways. Similar to a federated architecture, the integrated architecture decouples each DAS from other DASs. Each DAS possesses a dedicated encapsulated virtual network. A virtual network is private for a DAS, i.e., other DASs cannot perceive or affect the exchanged messages other than those being explicitly exported via a gateway. By exercising this strict control over the interactions between DASs, only the behavior of the DAS's virtual network and the behavior of gateways is of relevance when reasoning about a DAS. The message transmissions on other virtual networks can be abstracted from. Similarly, each job executes in a corresponding partition, which forms an encapsulated execution environment with guaranteed communication and computational resources. The activities of jobs executing on the same ECU cannot affect the resources that are available to other jobs in the ECU.

By not only supporting error containment between DASs, but error containment between jobs within a DAS, the integrated architecture exceeds the error containment capabilities of most federated systems. Furthermore, the integrated architecture offers generic architectural services as a base line for application development, thus decreasing the functionality that must be realized at the application level. Such a slimmer application is easier to comprehend and leaves less room for software design faults. Only the interface between the architecture and the application is visible to the application developer, while the realization of the architectural services remains hidden.

## Dependability and Mixed Criticality Integration

Carmakers are on the verge of deploying by-wire technology to improve the functionality of the vehicle that goes beyond traditional hydraulic and mechanic systems. This trend requires the deployed E/E architectures to meet the requirement for ultra-high dependability [SWH95] (a maximum failure rate of  $10^{-9}$  critical failures per hour is demanded). In the proposed architecture, the support for applications up to the highest criticality classes ( $10^{-9}$  failures per hour) is based on the following four properties:

**1. Replica determinism to support TMR.** Since ECU failure rates are in the order of  $10^{-5}$  to  $10^{-6}$ , ultra-dependable applications require the system as a whole to be more reliable than any one of its ECUs. This can only be achieved by utilizing fault-tolerance strategies that enable the continued operation of the system in the presence of ECU failures [BJV91]. The consequence of a hardware fault will in general be a complete failure of an ECU with all the jobs located on the ECU [OP06]. For this reason, it is necessary to use replication (e.g., TMR) where the replicated jobs are assigned to different ECUs<sup>1</sup>.

In order to support the implementation of TMR, the proposed architecture provides replica determinism [Pol94]. Replica determinism ensures that a majority decision with exact voting can be performed upon the outputs of three replicated jobs on different ECUs (without costly agreement protocols). Among the key mechanisms of the architecture for the establishment of replica determinism are the support for time-triggered control and static resource allocations. For example, the time-triggered virtual networks preclude race conditions in the access to the communication resources by design.

<sup>1</sup>This strategy does not preclude, however, the ability to integrated multiple jobs on a single ECU as long as the jobs are not part of the same TMR configuration.

**2. Different classes of high-level architectural services.** We distinguish safety-critical and non safety-critical DASs. The non safety-critical DASs possess only benign failure modes [Lap92], while safety-critical DASs exhibit critical failure modes that can lead to catastrophes and an endangerment of human life. Along with the distinction of safety-critical and non safety-critical DASs, the proposed architecture performs a likewise differentiation of the high-level architectural services. For the safety-critical DASs, the design of the high-level architectural services must be as simple as possible (e.g., only time-triggered virtual networks) in order to facilitate formal analysis and certification. For the non safety-critical DASs, on the other hand, high-level architectural services with additional functionality can be provided to ease the application development. For example, event-triggered virtual networks improve flexibility and resource efficiency through support for on-demand message exchanges. Furthermore, the integration of legacy applications without redevelopment efforts can require functionality for emulating the corresponding legacy platforms (e.g., emulation of CAN). This increased complexity is tolerable, when certification to the highest criticality levels (e.g., SIL4 in EN ISO/IEC 61508 [IEC99]) is not required.

**3. Error containment between ECUs enforced by core architecture.** The time-triggered core architecture uses a TDMA communication scheme, in which each communication slot possesses a unique sender ECU. The proposed architecture exploits bus guardians, which are available for several time-triggered networks (e.g., FlexRay, TTA), to protect these communication slots. The error containment between ECUs is significant to prevent common mode failures of the replicas in a TMR configuration.

**4. Error containment within an ECU enforced by high-level architectural services.** On top of the error containment mechanisms of the core architecture, the proposed architecture prevents an error of a faulty job to propagate to other jobs on the same ECU. The error containment within an ECU occurs in two ways:

- **Partition management operating system.** The partition management operating system (cf. Section 6) protects the computational resources by means of temporal and spatial partitioning. Temporal partitioning uses a scheduler that ensures each job gets its guaranteed CPU time, e.g., using a fixed-cyclic time-slice scheduler.
- **Virtual networks.** The virtual networks (cf. Section 4) protect the communication resources based on the hierarchic subdivision of the communication slots in the TDMA scheme. Each communication slot contains messages from a single job only. Thus, guardians can prevent a job from sending in a communication slot that belongs to another job.

This extended error containment is of particular importance for mixed criticality systems, in which jobs of safety-critical and non safety-critical DASs coexist on the same ECU. In general, such DASs will exhibit significant differences concerning the residue of design faults after deployment due to different development processes driven by economic constraints. In safety-critical application subsystems, the absence of design faults can no longer be shown by testing alone. The achievement of the reliability includes a rigorous development process, formal verification, and involvement of a certification agency. For non safety-critical application subsystems, certainty about the complete absence of design faults is usually economically infeasible. The level of rigidity in the development process of safety-critical applications would be too expensive.

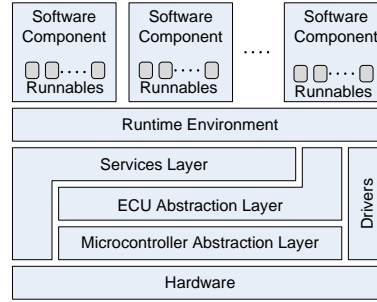


Figure 8. ECU in AUTOSAR

## Flexibility

A newly developed E/E architectures is expected to be deployable on different car segments and models. Consequently, the reuse of applications in a modular way is of critical importance. This issue also has an impact on “Tier 1 system suppliers” that need to adapt their subsystem design according to this trend. The usage of validated and possibly certified application software in combination with standard physical ECUs on different models and segments of vehicles will lead to advantages in terms of increased volume, multi-supplier management, and multi-platform management. In addition, not only software modules but complete electronic subsystems can be made available for different vehicle platforms. This strategy also eases the design choices for the interior of the car, due to the freedom of decoupling application software from a particular ECU. The integrated automotive architecture provides a high degree of freedom in the allocation of jobs to ECUs and supports the migration of jobs between ECUs (constrained by dependability requirements).

## 7. Related Work

This section gives an overview of related work on integrated architectures (AUTOSAR and IMA). We focus on the sharing of the computational resources via ECUs with support for multiple software components. In addition, the sharing of communication resources is addressed. The section also discusses the relationship to the integrated architecture proposed in this paper.

### AUTOSAR

The AUTOSAR [AUT06b] is a system architecture and development methodology for automotive electronic systems. Among the primary goals of AUTOSAR are the standardization of the basic software of an ECU (called standard core) and the ability to integrate software components from multiple suppliers. Another focus of AUTOSAR is the establishment of portability and location transparency for software components in order to accommodate future changes to the automotive electronic systems and facilitate reuse across product lines [Rol06].

**Model of an ECU.** An ECU in AUTOSAR supports the integration of software components from multiple suppliers (see Figure 8). A *software component* is a piece of software that can be mapped to an ECU. Internally, a software component contains threads of control called *runnables*. AUTOSAR distinguishes between event-triggered and communication-triggered runnables. The execution of event-triggered runnables is triggered by the occurrence of a significant event

(e.g., timeout). Communication-triggered runnables are activated by the arrival of messages from other software components.

The *runtime environment* decouples the software components from each other and from the hardware. It provides mechanisms for communication between the software components in the same ECU and on different ECUs.

Below the runtime environment in Figure 8, the AUTOSAR ECU contains the basic software encompassing layers with standardized functionality. These layers consist of software components that do not fulfill application functionality themselves, but interact with the application software components using the runtime environment.

- **Services layer.** This layer provides operating system services (e.g., priority-based scheduling of runnables, memory management), and communication services (e.g., communication stack for CAN or LIN).
- **ECU abstraction layer.** For accessing peripherals and devices, this layer establishes an API that abstracts from the underlying hardware. For example, the I/O hardware and the communication hardware (e.g., CAN controller) deployed on the ECU are hidden from the upper layers.
- **Microcontroller abstraction layer.** This layer provides access to internal and external peripherals of the ECU.
- **Drivers.** The drivers are application-dependent and interact with sensors and actuators. In order to satisfy tight timing requirements (e.g., injection control), the drivers can use interrupts and directly access the microcontroller and the peripherals.

**Communication System.** For interactions between software components, AUTOSAR provides a so-called *virtual function bus* with two communication patterns:

- The *sender-receiver communication* supports the transmission of signals with atomic data elements from a sending software component to one or more receiving software components. A sender-receiver communication is unidirectional, i.e., a reply involves another sender-receiver communication.
- The *client-server communication* enables a software component (acting as the client) to invoke a server function at another software component. The function invocation by the client can be synchronous (i.e., blocking until the result arrives from the server) or asynchronous (i.e., non-blocking).

The implementation of the virtual function bus occurs using the runtime environment, which acts as a communication switch and ensures location transparency. Communication between software components on same ECU is realized by passing arguments directly to the respective runnables. Communication between software components on different ECUs exploits the services layer of the ECU in order to establish a mapping to the communication network (e.g., CAN).

**Relationship to the DECOS Architecture.** Like the architecture proposed in this paper, AUTOSAR aims at evolving from today's ECU-centric approach to a development process that models application subsystems via software components. Both architectures support the integration of software components from multiple suppliers on shared ECUs, while providing architectural services as a baseline for the application development.



The contributions of the architecture proposed in this paper, which go beyond the features of AUTOSAR, include the following properties:

- **Rigorous encapsulation and error containment.** Based on static resource allocations at the communication network (i.e., TDMA scheme) and the operating system (i.e., fixed cyclic scheduler), the proposed architecture ensures that a job cannot affect the computational and communication resources that are available to other jobs. As discussed in Section 6, this property is of particular significance for the modular certification of mixed criticality systems, the implementation of active redundancy, and complexity management.
- **Predictability and determinism.** The rigorous encapsulation simplifies a timing analysis of application subsystems and jobs. When determining the temporal behavior of a job, a designer can abstract from the behavior of other jobs, because the architecture guarantees to each job predefined communication and computational resources.

In contrast, the current version of the AUTOSAR runtime environment [AUT06a] supports priority-based mechanisms for communication and the scheduling of runnables. For example, a high-priority runnable of one software component could delay the execution of runnables of other software components. In general, it is thus necessary to analyze the execution behavior of all software components with higher priority on an ECU. Similar dependencies between software components can occur at the communication system when using a priority-based protocol such as CAN.

- **Time-triggered core architecture.** The proposed architecture builds on top of a time-triggered network (e.g., FlexRay), the services of which (e.g., fault isolation, time-triggered message transport) are the foundation for rigorous encapsulation and the achievement of sufficient dependability for safety-critical applications (e.g., X-by-wire). AUTOSAR, on the other hand, does not enforce this restriction. The standard core with its runtime environment supports event-triggered protocols such as CAN for coupling software components.

Nevertheless, these contributions do not contradict the AUTOSAR development methodology. They can also be used to improve AUTOSAR, e.g., through better complexity management and support for safety-critical applications. This includes the mechanisms for encapsulation and error containment, as well as the mapping to an underlying time-triggered core architecture with high predictability and determinism.

## Integrated Modular Avionics

While present day automotive electronic systems adhere to the federated architectural paradigm, integrated architectures have already been successfully applied in the development of the electronic systems of commercial aircrafts (e.g., 777 [Wit96]). ARINC standard 651 [Aer91] is known as Integrated Modular Avionics (IMA) and addresses the design of integrated avionic systems. The construction of an IMA architecture relies on several other ARINC standards. For instance, the services of the avionic software environment are specified by ARINC 653 [Aer06], which is known as APplication EXecutive (APEX). APEX provides services for partition management, process management, time management, memory management, interpartition communication, intrapartition communication, and diagnosis. IMA systems are not restricted to a particular communication network. Common standards for the communication network include event-triggered communication systems (e.g., ARINC 664 [ARI03]), and time-triggered communication systems (e.g., ARINC 659 – SAFEbus [ARI93]).

**Model of an ECU.** ARINC specification 653 distinguishes between *core software* and *application software*. The core software is responsible for mapping the APEX API onto the underlying platform, e.g., implementing channels through the available transport mechanisms. If the core software employs fragmentation, sequencing, routing, or message redundancy, these mechanisms have to be transparent to the application software.

Each end-system contains one or more *partitions*. A partition is a set of functionally separated tasks with their associated context and configuration data. The tasks of a partition and the required resources are statically defined. The scheduling of these tasks occurs via two-level scheduling. On the first level, partitions are scheduled by the progression of time. Partitions do not have priorities and are activated relative to a time frame. As the time frame can be synchronized to the underlying communication system, the activation of partitions can also be synchronized to the communication system. Tasks within partitions possess priorities, which are used for dynamic task scheduling. If a partition is active, the scheduler selects the ready task with the highest priority for execution – preempting task's with lower priorities.

**Communication System.** APEX defines *channels* for interpartition communication through the exchange of messages. The destination for messages exchanged through a channel is a partition, not a process. Communication activities are independent of the physical location of both source and destination partitions. A channel is configured by the system integrator and possesses exactly one sending port and one or more receiving ports. Each port is assigned a port name, which should refer to the data exchanged via the port rather than to the producer/consumer. A port can support either event or state semantics through operating in one of the following two transfer modes:

- **Sampling Mode:** Successive messages contain identical but updated data. Received messages overwrite old information, thus requiring no message queuing. Sampling mode assumes that applications are only interested in the most recent version of a message. A validity indicator denotes whether the age of the copied message is consistent with the required refresh rate defined for the port.
- **Queuing Mode:** Messages are assumed to contain uniquely different data, thus no message should be lost. Messages are buffered in queues, which are managed on a First-In/First-Out (FIFO) basis. The application software is responsible for handling overflowing queues.

The concept of channels, as used for interpartition communication according to APEX, is independent of the actual transport mechanism. While a time-triggered communication system simplifies the establishment of temporal validity in sampling mode, event-triggered communication systems natively support sporadic transmissions of event messages as required in queuing mode.

**Relationship to the DECOS Architecture.** The DECOS integrated architecture and IMA/APEX share the vision of a physically integrated computer system, in which communication resources as well as computational resources are available to multiple applications. In particular, both architectures emphasize the need for encapsulating communication resources and computational resources (CPU, memory) in order to prevent interference induced by physical integration. However, we can identify the following properties, in which IMA and the integrated DECOS architecture differ:

- **Domain-independence.** While DECOS aims at offering a domain-independent architecture, IMA focuses explicitly on the avionic domain. For example, the automotive domain

significantly differs with respect to life cycles, cost sensitivity of end products, and requirements with respect to flexibility. The automotive domain exhibits a higher number of variants and mass customization w.r.t. electronic systems plays an important role. The individualizing of a car to its customer has a high impact on the electronic systems.

- **System Structuring.** IMA/APEX adheres to a “partition-centric” point of view, while DECOS adopts an “application subsystem”-centric viewpoint. IMA/APEX does not support within the integrated system a set of “virtualized” clusters, as they would exist in a federated system. However, part of the success of the federated paradigm is the ability to structure the overall system into nearly-independent subsystems, each becoming the responsibility of a respective organizational entity (e.g., a supplier). Following this line of reasoning, DECOS supports within a physically integrated computer system a similar logical structuring as a federated computer system. This decoupling of the physical system structuring from the logical system structuring is the key element to achieve the benefits of federated and integrated computer systems. In adopting an application-subsystem centric viewpoint, in DECOS each DAS is provided with a corresponding set of architectural services (e.g., communication services, diagnostic services, time services, gateway services).
- **Communication Service.** IMA provides to partitions support for communication via queueing mode and sampling mode, a distinction which corresponds to the event-triggered and time-triggered control paradigms used in this paper. At the DAS-level, however, DECOS goes beyond the establishment of communication channels for event and state messages, but realizes virtual networks running different communication protocols as encapsulated overlay networks. The underlying design assumption is that at the level of a DAS, the developer is in a position to perform a meaningful decision on a particular communication protocol that best suits the requirements of the DAS (e.g., balanced compromise between conflicting properties, such as flexibility vs. predictability).
- **Generic Gateway Service.** In contrast to IMA, the DECOS architecture provides a generic architectural gateway service that can be parameterized to specific applications. Gateways support the selective redirection of messages between the virtual networks of different DASs. Gateways make dependencies between DASs explicit and provide a systematic solution for resolving property mismatches (e.g., differences w.r.t. syntax, protocols, or naming). Within a gateway, a real-time data base separates the DASs and stores temporally accurate real-time images.

## 8. Conclusion

Future car generations require computer architectures to accommodate the need for mixed criticality applications, i.e., supporting applications with ultra-high dependability requirements as well as applications where flexibility and resource efficiency is of primary concern (e.g., comfort electronics). The proposed architecture establishes such an infrastructure and also enables physical integration by combining multiple DASs and virtual networks within a single distributed real-time computer system. Thus, the architecture reduces the number of different networks and protocols.

The proposed integrated architecture exhibits flexibility and supports reuse of application software across different car segments. The key element for this flexibility, as well as for complexity management and the independent development of subsystems, are small DASs. Instead of the typical domain oriented system structure, we show that we can subdivide the overall functionality of a car into smaller DASs, each equipped with dedicated architectural services. By transforming

a today's automotive system onto the future E/E architecture, we have demonstrated the feasibility of the integrated architecture for a future automotive system.

## Acknowledgments

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

## References

- [ARI91] Aeronautical Radio, Inc., Annapolis, Maryland 21401. *ARINC Specification 651: Design Guide for Integrated Modular Avionics*, November 1991.
- [ARI93] Aeronautical Radio, Inc., Annapolis, Maryland 21401. *ARINC Specification 659: Backplane Data Bus*, December 1993.
- [ARI03] Aeronautical Radio, Inc., Annapolis, Maryland 21401. *ARINC Specification 664 (Draft): Aircraft Data Network Part 7 – Deterministic Networks*, May 2003.
- [ARI06] Aeronautical Radio, Inc., Annapolis, Maryland 21401. *ARINC Specification 653: Avionics Application Software Standard Interface, Part 1 - Required Services*, March 2006.
- [ASBT03] A. Ademaj, H. Sivencrona, G. Bauer, and J. Torin. Evaluation of fault handling of the time-triggered architecture with bus and star topology. In *Proc. of the 2003 Int. Conference on Dependable Systems and Networks*, pages 123–132, June 2003.
- [AUT06a] AUTOSAR GbR. *AUTOSAR – Specification of RTE Software V1.0.1*, July 2006.
- [AUT06b] AUTOSAR GbR. *AUTOSAR – Technical Overview V2.0.1*, June 2006.
- [BJV91] R.W. Butler, J.L. Caldwell, and B.L. Di Vito. Design strategy for a formally verified reliable computing platform. In *Proc. of the 6th Annual Conference on Computer Assurance Systems*, pages 125–133, June 1991.
- [BKS03] G. Bauer, H. Kopetz, and W. Steiner. The central guardian approach to enforce fault isolation in a time-triggered system. In *Proc. of the 6th Int. Symposium on Autonomous Decentralized Systems*, pages 37–44, April 2003.
- [Bos91] Robert Bosch GmbH, Stuttgart, Germany. *CAN Specification, Version 2.0*, 1991.
- [BS05] B. Bouyssounouse and J. Sifakis, editors. *Embedded Systems Design*. Springer Verlag, 2005.
- [C. 02] C. Jones et al. Final version of the DSoS conceptual model. *DSoS Project (IST-1999-11585)*, December 2002.
- [Elm02] Wilfried Elmenreich. *Sensor Fusion in Time-Triggered Systems*. PhD thesis, Technische Universitat Wien, Institut für Technische Informatik, 2002.
- [Fle05] FlexRay Consortium. *FlexRay Communications System Protocol Specification Version 2.1*, May 2005.
- [GFL<sup>+</sup>02] P. Giusto, A. Ferrari, L. Lavagno, J.-Y. Brunel, E. Fourgeau, and A. Sangiovanni-Vincentelli. Automotive virtual integration platforms: why's, what's, and how's. In *Proc. of the IEEE Int. Conference on Computer Design: VLSI in Computers and Processors*, pages 370–378, September 2002.
- [Ham03] R. Hammett. Flight-critical distributed systems: design considerations [avionics]. *IEEE Aerospace and Electronic Systems Magazine*, 18(6):30–36, June 2003.
- [HB98] B. Hedenetz and R. Belschner. Brake-by-wire without mechanical backup by using a TTP-communication network. In *Proceedings of SAE Congress*. Daimler-Benz AG, 1998.
- [He04] H. Heinecke and et al. AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures. In *Proc. of the Convergence Int. Congress & Exposition On Transportation Electronics*, October 2004. 2004-21-0042.
- [Hei03] H.D. Heitzer. Development of a fault-tolerant steer-by-wire steering system. *Auto Technology*, 4:56–60, April 2003.
- [HPOS05] B. Huber, P. Peti, R. Obermaisser, and C. El Salloum. Using RTAI/LXRT for partitioning in a prototype implementation of the DECOS architecture. In *Proc. of the Third Int. Workshop on Intelligent Solutions in Embedded Systems*, May 2005.

- [IEC99] IEC: Int. Electrotechnical Commission. *IEC 61508-7: Functional Safety of Electrical, Electronic, Programmable Electronic Safety-Related Systems – Part 7: Overview of Techniques and Measures*, 1999.
- [JB92] S.C. Johnson and R.W. Butler. Design for validation. *IEEE Aerospace and Electronic Systems Magazine*, 7(1):38–43, January 1992.
- [KAGS05] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The Time-Triggered Ethernet (TTE) design. *Proc. of 8th IEEE Int. Symposium on Object-oriented Real-time distributed Computing (ISORC)*, May 2005.
- [KB03] H. Kopetz and G. Bauer. The time-triggered architecture. *IEEE Special Issue on Modeling and Design of Embedded Software*, January 2003.
- [KO02] H. Kopetz and R. Obermaisser. Temporal composability. *Computing & Control Engineering Journal*, 13:156–162, August 2002.
- [Kop97] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.
- [Kop03] H. Kopetz. Fault containment and error detection in the time-triggered architecture. In *Proc. of the Sixth Int. Symposium on Autonomous Decentralized Systems*, April 2003.
- [KS03] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *Proc. of the 6th IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing*, pages 51–60, May 2003.
- [KWR04] L. Kinnan, J. Wlad, and P. Rogers. Porting applications to an ARINC 653 compliant IMA platform using VxWorks as an example. In *Proc. of the 23rd Digital Avionics Systems Conference*, volume 2, pages 10.B.1–10.1–8, October 2004.
- [Lap92] J.C. Laprie. Dependability: Basic concepts and terminology. In *Dependable Computing and Fault Tolerant Systems*, volume 5, pages 257–282. Springer Verlag, Vienna, 1992.
- [Lev86] N.G. Leveson. Software safety: why, what, and how. *ACM Comput. Surv.*, 18(2):125–163, 1986.
- [LH94] J.H. Lala and R.E. Harper. Architectural principles for safety-critical real-time applications. *Proc. of the IEEE*, 82:25–40, January 1994.
- [LH02] G. Leen and D. Heffernan. Expanding automotive electronic systems. *Computer*, 35(1):88–93, January 2002.
- [LIN03] LIN Consortium. *LIN Specification Package Revision 2.0*, September 2003.
- [Lyn06] LinuxWorks. *LynxOS 4.0 User's Guide*, 2006.
- [Nex03] NextTTA. Project deliverable D2.4. Emulation of CAN and TCP/IP, September 2003. IST-2001-32111. High Confidence Architecture for Distributed Control Applications.
- [OP05a] R. Obermaisser and P. Peti. Comparison of the temporal performance of physical and virtual CAN networks. In *Proc. of the IEEE Int. Symposium on Industrial Electronics*, Dubrovnik, Croatia, June 2005.
- [OP05b] R. Obermaisser and P. Peti. Realization of virtual networks in the DECOS integrated architecture. In *Proc. of the Workshop on Parallel and Distributed Real-Time Systems 2006 (WPDRTS)*. IEEE, April 2005.
- [OP05c] R. Obermaisser and P. Peti. Specification and execution of gateways in integrated architectures. In *Proc. of the 10th IEEE Int. Conference on Emerging Technologies and Factory Automation*, Catania, Italy, September 2005. IEEE.
- [OP06] R. Obermaisser and P. Peti. A fault hypothesis for integrated architectures. In *Proc. of the 4th Int. Workshop on Intelligent Solutions in Embedded Systems*, June 2006.
- [OPHS06] R. Obermaisser, P. Peti, B. Huber, and C. El Salloum. DECOS: An integrated time-triggered architecture. *e&i journal (journal of the Austrian professional institution for electrical and information engineering)*, 3:83–95, March 2006.
- [PO06] P. Peti and R. Obermaisser. A diagnostic framework for integrated time-triggered architectures. In *Proc. of the 9th IEEE Int. Symposium on Object-oriented Real-time distributed Computing*, April 2006.
- [Pol94] S. Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6:289–316, 1994.
- [Rei06] K. Reif. *Automobilelektronik. Eine Einführung für Ingenieure (Broschiert)*. Vieweg, May 2006.
- [RH04] G. Reichart and M. Haneberg. Key drivers for a future system architecture in vehicles. In *Convergence Int. Congress*. SAE, October 2004.

- [Rol06] T. Rolina. Past, present, and future of real-time embedded automotive software: A close look at basic concepts of AUTOSAR. In *Proc. of SAE World Congress*, Detroit, Michigan, April 2006.
- [Rus01] J. Rushby. Bus architectures for safety-critical embedded systems. In *Proc. of the 1st Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 306–323. Springer-Verlag, October 2001.
- [Sim96] H.A. Simon. *The Sciences of the Artificial*. MIT Press, 1996.
- [SLO03] M. Segarra, T. Losert, and R. Obermaisser. Hard real-time CORBA: TTP transport definition. Technical Report IST37652/067, Universidad Politecnica de Madrid, March 2003.
- [SM99] J. Swingler and J.W. McBride. The degradation of road tested automotive connectors. In *Proc. of the 45th IEEE Holm Conference on Electrical Contacts*, pages 146–152, October 1999.
- [SW04] A. Saad and U. Weinmann. Intelligent automotive system services: Requirements, architectures and implementation issues. In *Convergence Int. Congress*, Detroit, MI, USA, October 2004. SAE.
- [SWH95] N. Suri, C.J. Walter, and M.M. Hugue. *Advances In Ultra-Dependable Distributed Systems*, chapter 1. IEEE Computer Society Press, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA 90720-1264, 1995.
- [TTT02] TTTech Computertechnik AG. *Time-Triggered Protocol TTP/C – High Level Specification Document*, July 2002.
- [Wit96] B. Witwer. Systems integration of the 777 airplane information management system (aims). *IEEE Aerospace and Electronic Systems Magazine*, 11(4):17–21, April 1996.

# TEMPORAL PARTITIONING OF COMMUNICATION RESOURCES IN AN INTEGRATED ARCHITECTURE

*IEEE Transactions on Dependable and Secure Computing, October 2007, IEEE Computer Society.*

Roman Obermaisser  
Vienna University of Technology  
Real-Time Systems Group  
romano@vmars.tuwien.ac.at

**Abstract** Integrated architectures in the automotive and avionic domain promise improved resource utilization and enable a better coordination of application subsystems compared to federated systems. An integrated architecture shares the system's communication resources by using a single physical network for exchanging messages of multiple application subsystems. Similarly, the computational resources (e.g., memory, CPU time) of each node computer are available to multiple software components. In order to support a seamless system integration without unintended side effects in such an integrated architecture, it is important to ensure that the software components do not interfere through the use of these shared resources. For this reason, the DECOS integrated architecture encapsulates application subsystems and their constituting software components. At the level of the communication system, virtual networks on top of an underlying time-triggered physical network exhibit predefined temporal properties (i.e., bandwidth, latency, latency jitter). Due to encapsulation the temporal properties of messages sent by a software component are independent from the behavior of other software components, in particular from those within other application subsystems. This paper presents the mechanisms for temporal partitioning of communication resources in the DECOS integrated architecture. Furthermore, experimental evidence is provided in order to demonstrate that the messages sent by one software component do not affect the temporal properties of messages exchanged by other software components. Rigid temporal partitioning is achievable, while at the same time meeting the performance requirements imposed by present-day automotive applications and those envisioned for the future (e.g., X-by-wire). For this purpose, we use an experimental framework with an implementation of virtual networks on top of a TDMA-controlled Ethernet network.

**Keywords:** System architectures, real-time and embedded systems, system integration and implementation, fault-tolerance, computer network performance, distributed architectures, infrastructure protection

## 1. Introduction

In application domains with large distributed embedded computer systems, such as the avionic or automotive domains, integrated architectures are viewed as superior compared to federated architectures. While federated architectures provide each application subsystem (e.g., powertrain subsystem in a car) with a dedicated set of nodes interconnected by a respective network, integrated architectures permit the coexistence of multiple application subsystems within a single distributed computer system. Integrated architectures support the sharing of computational resources (i.e., node computers with corresponding CPU time, memory, I/O) and communication resources

(i.e., network bandwidth) among multiple software components belonging to different application subsystems. IMA [Aer91], AUTOSAR [ea04], and the cross-domain DECOS architecture [OP06] are examples of integrated architectures that are already partly deployed (IMA) or still in development (AUTOSAR, DECOS). Especially in the automotive domain, integrated architectures are a solution that promises to stop the continuous increase in the number of node computers, which has been a side-effect of the increasing functionality realized by electronic systems in the past years. While significant advances w.r.t. safety and comfort can be attributed to electronics, the high number of node computers and networks has resulted in high hardware cost, weight, and power consumption. For example, with an average cost of 40 Euro per node computer in today's cars and 0.2 Euro per wire, the total hardware cost of a federated automotive system with 40 nodes and 800 wires is about 1800 Euro [POT<sup>+</sup>05]. In addition, increased wiring can affect the reliability of automotive electronics. Field data from automotive environments has shown that more than 30% of electrical failures are attributed to connector problems [SM99]. Even so, the functionality of electronic systems is expected to continue its rapid growth. Automotive electronics are estimated to enable 90% of all innovations in cars [BS05][p. 15].

Despite the importance of deploying integrated architectures, it is important to recognize the benefits that have resulted from the natural separation of application subsystems in federated architectures. Each application subsystem possesses a functional complexity that is inherent to the application. The functional complexity of an application subsystem when implemented on a target system is dramatically increased by the so-called accidental complexity [Bro87] in case the architecture does not prevent unintended architecture-induced side effects in the communication system. For example, consider a scenario with two application subsystems. If the two application subsystems share a common CAN bus [Bos91], then both application subsystems must be analyzed and understood in order to reason about the correct behavior of either of the two application subsystems. Since the message transmissions of one application subsystem can delay message transmission of the other application subsystem, arguments concerning the correct temporal behavior must be based on an analysis of both application subsystems.

The challenge for the communication infrastructure of an integrated architecture is therefore the prevention of accidental complexity through architecture-induced side effects in the communication system. In analogy to a federated architecture, the communication system of an integrated architecture must support the division of the overall functionality into separate application subsystems with guaranteed communication resources. When constructing mixed criticality systems, which are composed of application subsystems with different criticality levels, it is also important to preserve these guarantees in the presence of design faults. Otherwise, error propagation to subsystems with higher criticality could occur through the communication system.

In previous work, we have developed the conceptual foundation for the communication infrastructure of the DECOS architecture along with a model-based development process [OH06]. Each application subsystem is provided with a dedicated communication infrastructure that is realized as an encapsulated *Virtual Network (VN)*, i.e., an overlay network on top of a time-triggered physical network. Each VN supports a corresponding communication paradigm (i.e., time-triggered or event-triggered control [Kop97]) and is tailored to the requirements of the respective application subsystem via its temporal properties (e.g., latencies, bandwidth) and its communication topology (e.g., broadcast, multicast, point-to-point).

The *encapsulation mechanisms* of VNs, which will be explained in this paper, address the challenge of preventing architecture-induced side effects in the communication system. By supporting temporal partitioning [Rus99] both between and within application subsystems, VNs not only help developers to focus on the inherent application complexity, but also contribute to the correctness-



by-construction of component-based systems (i.e., composability [Sif05]). The temporal properties of a VN remain invariant throughout the system integration. This property called *temporal composability* [KO02] is an important baseline for a constructive system development that prevents unintended interference between independently developed software components. Furthermore, encapsulation also addresses the integration of mixed criticality systems. Encapsulation ensures that a failure of a software component belonging to a non safety-critical application subsystem does not affect the exchange of messages between software components of safety-critical application subsystems.

The paper is a substantial extension of previous work in the context of the DECOS architecture w.r.t. virtual networks [OPK05, OH06] and operating systems [HPOS05]. While [HPOS05] has focused on the temporal partitioning (i.e., CPU time) and spatial partitioning (i.e., memory protection) for the *computational resources* [HPOS05], the present paper describes the partitioning for the *communication resources* (i.e., communication bandwidth, latency, jitter). In extension to previous work on virtual networks [OPK05, OH06], we introduce mechanisms for temporal partitioning of overlay networks on top of a time-triggered network and perform an experimental evaluation. We provide experimental evidence for the effectiveness of temporal partitioning in a prototype implementation with multiple event-triggered and time-triggered overlay networks. Despite rigid temporal partitioning, the measurements also show that the overlay networks can handle the performance requirements of today's automotive electronic systems and those of future X-by-wire cars.

The main contributions of this paper include:

- **Mechanisms for temporal partitioning in the communication system of an integrated architecture.** We present a conceptual model of an integrated computer system that distinguishes clearly between logical and physical structuring. Based on this model, we use the communication slots of a time-triggered physical network and subdivide them hierarchically for the structural entities of the logical and physical system structuring. Software mechanisms (e.g., communication middleware) in conjunction with hardware mechanism (e.g., bus guardians) protect these communication slots down to the level of individual software components, which can be collocated on shared integrated node computers.
- **Communication infrastructure for heterogeneous application subsystems.** The presented communication system supports both time-triggered and event-triggered communication activities and the coexistence of application subsystems with mixed criticality levels.
- **Experimental assessment of temporal partitioning.** In this paper, the invariance of the temporal properties of a communication system comprising multiple VNs is subject to comprehensive tests. We provide experimental evidence for the guaranteed temporal properties of the message exchanges. Two experimental campaigns systematically explore different scenarios for the behavior of software components at the communication system. We also assess the effects of faulty software components (e.g., babbling idiot failures).
- **Experimental assessment of performance.** By comparing the observed performance with the bandwidth and latency requirements of present day and upcoming automotive applications, we demonstrate that a communication system with rigid temporal partitioning can also support a competitive temporal performance.

## 2. Basic Concepts and Related Work

Partitioning is concerned with ensuring that a failure in one fault-containment region does not propagate to cause a failure in another fault-containment region. Effective partitioning mechanisms are thus important in order to preserve the independence of fault-containment regions and prevent common mode failures. Temporal partitioning [Rus99] is an instantiation of the general notion of partitioning and ensures that a fault-containment region cannot affect the ability of other fault-containment regions to access shared resources, such as the common network or a shared CPU. This includes the temporal behavior of the services provided by resources (e.g., latency, jitter, duration of availability during a scheduled access).

In integrated architectures, one can distinguish two types of fault-containment regions and two respective types of partitioning [OP06]. For physical faults, a complete node computer can be regarded as a fault-containment region. It is usually not justified to assume independent fault-containment regions on a single node computer due to the shared physical resources (e.g., power supply, oscillator, physical proximity). For design faults, on the other hand, each of the software components (that are collocated on a node computer) can be regarded as an individual fault-containment region. The reason for this finer structuring w.r.t. design faults in an integrated architecture is that software components on a node computer are typically self-contained functional elements, which are developed by different organizations.

Along with this discrimination of fault-containment regions, one can distinguish partitioning between nodes (i.e., *inter-node partitioning*) and partitioning between software components within a node (i.e., *inner-node partitioning*). In the following, we will give an overview of related work for these different types of temporal partitioning with a focus on time-triggered solutions. Also, solutions for mediating data flows between different levels of criticality will be addressed.

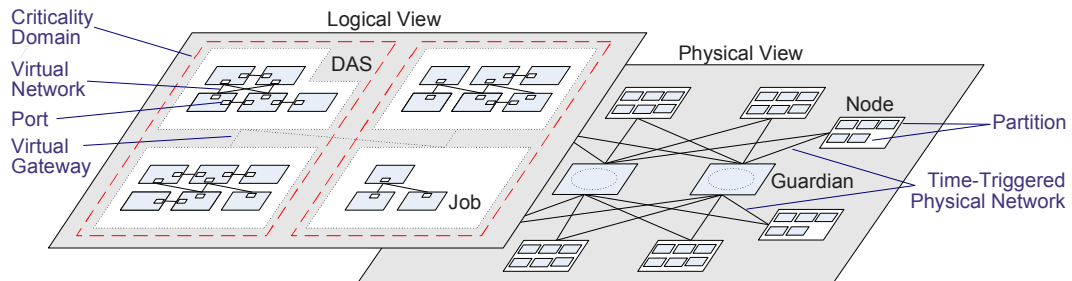


Figure 1. Modularization and Abstraction – Logical and Physical View

### Inter-Node Partitioning

For federated architectures that are deployed on time-triggered networks, temporal partitioning between nodes has been investigated extensively in previous work. The a priori knowledge about the delimiting points in time of a node's slot in the time-triggered communication scheme has been exploited by guardians that manage access to the underlying network in order to prevent a node from sending in the slot of another node. For this purpose, node-local and centralized guardians have been developed and validated for different time-triggered communication protocols (e.g., [BKS03, Gua05]).

### Inner-Node Partitioning for Computational Resources.

At the level of the computational resources, the primary source of temporal fault propagation within a node is a task that delays other tasks by holding a shared resource (e.g., the processor). Based on static scheduling of the computational resources, solutions for partitioning have been developed in the context of IMA. For example, LynxOS-178<sup>1</sup> is a certifiable operating system that supports temporal partitioning through a fixed-cyclic time-slice scheduler. Fixed-cyclic time-slice schedulers have also been combined with other scheduling policies using two-level hierarchical scheduling. For instance, [KLY00] proposes a microkernel with a cyclic/priority driven scheduler pair that guarantees temporal partitioning. Another example of a solution for temporal partitioning at the level of the computational resources is the formally verified DEOS scheduler kernel [ea00], which enforces time partitioning using a rate monotonic scheduling policy.

### Inner-Node Partitioning for Communication Resources.

In avionics, the need for temporal partitioning of communication resources within a line-replacable unit has gained high recognition with the introduction of IMA. When discussing the use of SAFEbus in IMA [HD93], it is stated that the execution environment of each function in a cabinet should be as much like the environment in the discrete line-replacable unit. Therefore, SAFEbus has been designed as a table-driven protocol, which enforces strict deterministic control for temporal partitioning between the line-replacable modules in a cabinet. However, line-replacable modules represent separate hardware elements equipped with dedicated hosts and bus interface units [HD93], although they typically share power supply and I/O units. In contrast, this paper addresses temporal partitioning w.r.t. communication resources at the level of software components within a node computer.

### Mediation of Data Flow between Different Levels of Criticality

The virtual networks presented in this paper provide temporal partitioning w.r.t. the communication resources. The only way in which a faulty job can affect other jobs is by providing to the other jobs faulty inputs. The elimination of interference in the use of communication resources is an important baseline for partitioning mechanisms at higher levels. In particular, higher levels can focus on the mediation of data flows between different levels of criticality. For example, many safety-critical control applications depend on sensory information from unreliable sources [TBDP98]. This criticality differential can be resolved by using fault-tolerance mechanisms and exploiting redundancy. For example, the GUARDS architecture proposes an information flow integrity policy [TBDP98] in which validation objects take low integrity inputs and run fault-tolerance mechanisms to produce high integrity outputs.

In the DECOS architecture, the mediation between data flows of different criticality is the purpose of so-called *virtual gateways* [OP05b]. Virtual gateways interconnect virtual networks, while enforcing error containment and resolving property mismatches. In addition, virtual gateways can recombine messages out of their constituting parts. For example, data contained in multiple messages from unreliable sources can be combined in order to send a message to a safety-critical application subsystem.

<sup>1</sup><http://www.linuxworks.com/>

### 3. System Model

*Modularization* and *abstraction* are two fundamental principles used for managing complexity [Par01]. Modularization deals with the meaningful decomposition of a system into smaller subsystems, while abstraction is concerned with the level of detail in the representation of a system or subsystem. This section discusses modularization and abstraction for a DECOS system, i.e., a system developed in compliance with the DECOS architecture, at both the logical and physical level (top and bottom levels in Figure 1). Furthermore, we introduce a naming scheme for identifying structural elements of the physical and logical level.

#### Logical View

At the top-most level of abstraction, a DECOS system (e.g., the complete on-board electronic system of a car) provides to its users (e.g., human operator) application services at the controlled object interface. By means of modularization, the DECOS system is structured into a set of nearly-decomposable *DASs*. Each DAS provides a subset of the overall application services, which is meaningful in the application context. An example for a DAS in the automotive domain would be a steer-by-wire subsystem [Hei03]. The DASs can be grouped into criticality domains, e.g., based on a common Safety Integrity Level (SIL) [IEC99] or critical failure modes [RTC92].

In analogy to the structuring of the overall system, we further decompose each DAS into smaller units called *jobs*. A *job* is the basic unit of work and employs a *Virtual Network (VN)* [OH06] with predefined temporal properties in order to exchange messages with other jobs of the DAS. The interface between a job and a VN is denoted as a *port*.

The two-level structuring into DASs and jobs is motivated by the insight that large distributed real-time systems typically consists of a set of DASs with high inner connectivity and looser interactions in-between. These different orders of magnitude in the interactions between subsystems correspond to the notion of near-decomposability in hierarchic systems as introduced in [Sim96, chap. 8]. For example, in the automotive domain a two-level structuring is applied to manage the complexity of the in-vehicle electronic system. In many present-day cars, the electronic system consists of several DASs (e.g., powertrain, comfort, passive safety subsystem), each of which is implemented with a distributed computer system. On its behalf, such a DAS is further structured into smaller logical elements that are implemented by electronic control units [Dei02, Leo04].

The DECOS integrated architecture facilitates the management of the complexity of large distributed real-time systems by enabling designers to perform a logical structuring of systems in the same way as if they were realizing DASs in a federated architecture. Although the DASs are physically integrated and the hardware (i.e., network, node computers) is shared, the DECOS integrated architecture encapsulates the elements of the logical system structuring (i.e., DASs, jobs) along with their communication resources (i.e., the VNs). Other DASs cannot perceive or affect exchanged messages (in the temporal or value domain) other than those being explicitly exported via a *virtual gateway* [OP05b]. By exercising this strict control over the interactions between DASs, only the behavior of the DAS's VN and the behavior of gateways is relevant when reasoning about a DAS.

#### Physical View

From a physical point of view (see bottom level in Figure 1), a DECOS system encompasses a cluster containing a set of *integrated node computers* (nodes for short), which are interconnected by a time-triggered physical network. The VNs introduced in the logical view are implemented

on top of this time-triggered physical network. The use of a time-triggered physical network matches the predictability and fault-tolerance requirements of safety-critical applications [Rus01]. Every node provides one or more *partitions*, each hosting a corresponding job. A partition is an encapsulated execution space within a node with a priori assigned computational (e.g., CPU, memory, I/O) and communication resources (e.g., network bandwidth, latencies). By supporting the deployment of multiple jobs on one node, the DECOS architecture goes beyond the prevalent “1 Function – 1 Electronic Control Unit” design principle [BS05].

## Architectural Namespace

The namespace of the integrated architecture, which has been introduced in [OPK05], consists of two parts, one reflecting the physical structure and one reflecting the logical structure of the system:

$$\underbrace{id_{\text{node}}}_{\text{physical structure}} : \underbrace{id_{\text{criticality}}.id_{\text{DAS}}.id_{\text{job}}}_{\text{logical structure}}$$

The physical part identifies the node ( $id_{\text{node}}$ ) of the integrated system, while the logical part following the colon identifies the criticality domain ( $id_{\text{criticality}}$ ), the DAS ( $id_{\text{DAS}}$ ), and the job ( $id_{\text{job}}$ ), where  $id$  is the numerical identification of a structuring element. Since each DAS possesses exactly one VN, the DAS identifier also designates its corresponding VN.

For convenience in writing a name, it is possible to omit either the physical or the logical part if not needed by the specification. For example, by describing only logical aspects, one can abstract from the physical allocation (i.e., specifying solely  $:id_{\text{criticality}}.id_{\text{DAS}}.id_{\text{job}}$ ).

## 4. Virtual Networks

An overlay network is a computer network which is built on top of another network. The DECOS architecture provides overlay networks, which are denoted as *Virtual Networks* (VNs), on top of a time-triggered physical network. Each VN handles the message exchanges of a corresponding DAS and provides encapsulation for the jobs by preventing jobs from affecting the temporal properties of messages sent by other jobs.

This section describes how the time-triggered physical network provides dedicated communication resources for the different structural elements introduced in the logical (i.e., criticality domains, DASs, jobs) and physical view-points (i.e., nodes). These communication resources are used to construct VNs for the exchange of event and state messages. In addition, we discuss the encapsulation of VNs based on the static resource allocation on the physical network and the design of the interfaces between VNs and jobs.

### Communication Resources provided by the Time-Triggered Physical Network

For the realization of the communication services in the integrated DECOS architecture, we employ a time-triggered physical network and perform a hierarchic temporal subdivision of the communication resources (see Figure 2). The media access control strategy of the time-triggered physical network is TDMA. TDMA statically divides the channel capacity into a number of slots and controls access to the network solely by the progression of time. Each node is assigned a unique *node slot* that periodically recurs at a priori specified global points in time. A node

sends messages during its node slot and receives messages during the slots of the other nodes. A sequence of node slots, which allows every node in an ensemble of nodes to send exactly once, is called a TDMA round.

We further subdivide each node slot (bottom-most layer in Figure 2) in correspondence to the logical structuring of a DECOS system. In a first step, the node slot is subdivided into subslots for the criticality domains. Thereafter, a subdivision into subslots for the DASs takes place. Such a *DAS slot* contains those messages that are produced by the jobs in the node that belong to a particular DAS. On its part, a DAS slot consists of smaller subslots denoted as *job slots* (top-most level in Figure 2). Each job slot provides the communication resources for the establishment of a *communication channel*, which serves the transport of messages produced by a particular sender job. A communication channel connects one output port (of the sender job) with multiple input ports (of the receiver jobs). The output port of a communication channel maps to exactly one job slot, namely the slot of the job possessing the output port. In this job slot the messages produced at the output port are sent on the underlying time-triggered network. The number of the input ports depends on the communication topology. For example, a single input port establishes a point-to-point topology, while a broadcast topology requires associated input ports for all other jobs of the DAS.

The composition of the communication channels for all jobs of a DAS results in a *VN*. In general, a *VN* comprises job slots that belong to jobs located on different nodes. In order to construct the *VN* of a DAS with multiple jobs that need to send messages, the *VN* requires a number of communication channels that is equal to the number of jobs in the DAS.

The assignment of the slots within a TDMA round to nodes, as well as the further subdivision into criticality-domain slots, DAS slots, and job slots is fixed at design time. This static allocation ensures that the network resources are predictably available to jobs. The definition of the TDMA scheme is an important design activity that determines the temporal properties of the resulting communication channels. Firstly, the period of the job slot in the TDMA scheme determines the latency for message transmissions that are not synchronized to the TDMA-scheme. In the worst case, a job writes a message into its output port after the occurrence of its job slot, thus incurring a delay of a complete TDMA round. Secondly, the size of the job slots in conjunction with the period of the job slots determines the available bandwidth.

## Virtual Networks for State and Event Messages

One can distinguish between state messages and event messages [Kop97, p. 31] depending on the temporal behavior and the information semantics. *State messages* are periodic messages with state information (e.g., temperature is 20°), while *event messages* are sporadic and contain event

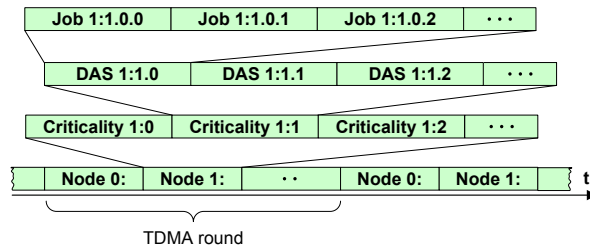


Figure 2. Hierarchic Subdivision of TDMA Slots

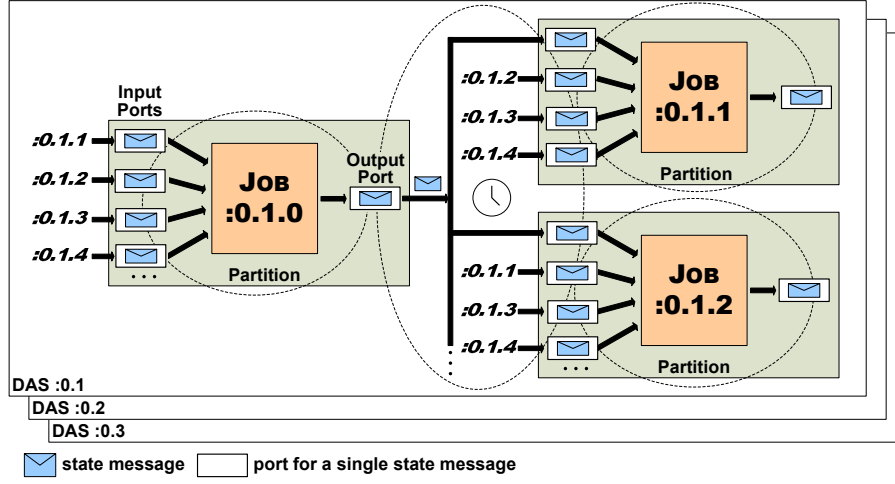


Figure 3. Virtual Network for Exchange of State Messages

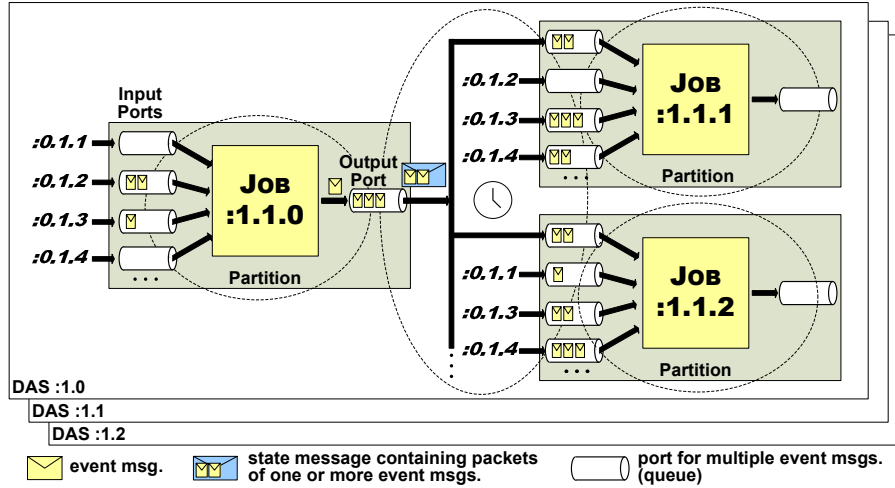


Figure 4. Virtual Network for Exchange of Event Messages

information (e.g., temperature increase by  $2^\circ$ ). Based on this distinction, there are two types of ports (event ports and state ports) as the interface between the application and the communication system. For any given communication channel, a particular port type must be used consistently.

**State ports.** A communication channel for the transport of state messages employs state ports, each containing a memory element that is overwritten with a new value whenever a state message arrives from the VN (in case of an input port) or the job (in case of an output port).

For example, Figure 3 depicts a VN for DAS :0.1 with communication channels for the exchange of state messages with a broadcast topology. Each of the jobs possesses a communication channel, which is interfaced by an output port with a memory element storing a state message. In addition, a job has an input port for every communication channel the job receives messages from. A message arriving via a communication channel causes an update of the memory element in the respective input port, but does not affect the contents of other input ports.

**Event ports.** An event port supports event messages containing information that is associated with a particular event. In order to reconstruct the current state of a real-time entity [Kop97, p. 98] from event messages, it is essential to process every message exactly once. Consequently, a VN must support the transmission of event messages with exactly-once delivery semantics and provide message queues at input and output ports. For the transport via the underlying time-triggered network, the communication channel packs event messages into state messages for the dissemination via the periodically reoccurring job slot.

Figure 4 depicts a VN for the exchange of event messages. In analogy to the previous example, this VN also establishes a broadcast topology. However, the interface between the jobs and the VN is provided with event ports.

Each port separates *two spheres of control*, namely the time-triggered control of a communication channel and the event-triggered control of a job (see dashed ellipses in Figures 3 and 4). At the communication channel, the transport of messages between the (one) output port and the input ports is periodically triggered by the occurrence of the sender's job slot in the TDMA scheme. Consequently, the communication channels employ strict time-triggered control. A job, on the other hand, can perform event-triggered access operations, i.e., at points in time that are not a priori known. A job acts according to the *information push paradigm* [DeL99] when updating a state message in the output state port or inserting an event message into an output event port. A job does not delegate control outside its sphere of control, but determines autonomously the point in time of an access operation. This job autonomy simplifies a timing analysis of the job and limits control error propagation. Likewise, the points in time for reading the memory element of an input port are chosen by the job, thus supporting message receptions according to the *information pull paradigm* [DeL99].

The design decision of supporting both state and event messages is motivated by the insight that both message types exhibit respective advantages. An integrated system can combine heterogeneous DASs with different requirements concerning the communication systems. While state messages are well-suited for safety-critical DASs (e.g., control applications), event messages are typically favorable in non safety-critical DASs with emphasis on flexibility. In the proposed architecture, event messages provide the following benefits:

- **Reuse of event-triggered legacy applications.** Event ports support the reuse of legacy applications that were designed to perform sporadic exchanges of event messages. For example, many CAN-based applications [Bos91] in today's cars send messages as a consequence of significant events in the environment (e.g., input from driver).
- **Message size is decoupled from size of job slots.** By using a packet service, communication channels for event messages decouple the size of transferred messages from the size of a job slot. For example, multiple small messages can be transmitted collectively within a single job slot or a large message can be fragmented over multiple slots within consecutive rounds.
- **Bandwidth elasticity.** The event ports of a communication channel for the exchange of event messages provide bandwidth elasticity. Due to the queues at the output ports, a job can pass more messages to the communication system than can be transmitted in a single TDMA round on the underlying time-triggered network. VNs can handle such a burst as long as the average bandwidth consumption can be bounded to dimension the job slots in the TDMA scheme, and the maximum message load of a burst is known in order to dimension the queue sizes.
- **Variability of a job's message service time.** The queue of an input port supports a variability of the message service time (i.e., the time between successive retrievals of messages through the job). However, in order to prevent queue overflows at the input port, the average service time



must be smaller than the average interarrival time of messages from the VN. In addition, a priori knowledge concerning message bursts is required to dimension the queue size.

- **Exactly-once semantics.** The connection between the output port at the sender and the input port at the receiver occurs with a communication channel that exhibits both unidirectional data flow and unidirectional control flow. Thereby, we prevent by design the introduction of an error propagation path from the receiver to the sender. While this design decision supports rigid temporal partitioning, it also implies restrictions for the exactly-once semantics. Exactly-once semantics only hold for a known message timing model, e.g., minimum message interarrival times with bursts of bounded duration and load. Such a message timing model can be used to dimension the queue lengths at input and output ports, as well as the size of the job slots.

For guaranteeing exactly-once semantics without sufficient a priori knowledge concerning the message timing, end-to-end control can be realized using higher protocols on top of a VN. For example, in [DEC06] such an end-to-end control scheme has been realized on top of a VN with support for event messages. However, the bidirectional control flow can introduce paths for error propagation from the receivers to the sender. Therefore, (at the basic level) the presented VNs adhere to the design decision of using only unidirectional communication channels, leaving the optional establishment of bidirectional control flows to higher protocols (e.g., using TCP/IP as in [DEC06]).

## Encapsulation of Virtual Networks

Due to encapsulation, developers need not look at all possible interactions between jobs and DASs in order to understand the temporal behavior of a VN. In particular, upon the occurrence of faults covered in the fault hypothesis [OP06], the encapsulation of VNs preserves the modularization of the overall system into DASs and jobs as introduced in the logical system structuring. The primary purpose of encapsulation is the prevention of adverse effects on the message exchanges of a particular VN induced by the message exchanges on VNs of other DASs. In addition to this encapsulation at the DAS-level, VNs are designed for encapsulation at the job-level. Encapsulation at the job-level encompasses the prevention of adverse effects on the message exchanges of a job induced by the message exchanges of other jobs in the same DAS.

Encapsulation confines the effects of a job failure that results in the transmissions of incorrect messages. In case of such a job failure, one can distinguish between message timing and message value failures. A message sent at an unspecified time is denoted as a *message timing failure*. Examples for specific message timing failures are crash/omission failures and babbling idiot failures [Cri91]. A *message value failure* occurs in case the contents of a transmitted message do not comply with the interface specification. In general, the detection of message value failure requires application-specific knowledge either through a priori knowledge or redundant computations. An example for the latter case is active redundancy (e.g., TMR), which supports the detection and masking of message value failures by majority voting. In the scope of this work, we focus on the encapsulation in the time domain by means of temporal partitioning.

For describing the temporal partitioning, we take on a *sender-centric view*. This means that we look at the non interference of the message transmissions between sender jobs, while abstracting over interference between message transmissions from the same sender job. The reason for this view is the fault hypothesis of the DECOS architecture [OP06], which regards each job as a distinct fault-containment region w.r.t. to software faults.

Providing a dedicated input port for each communication channel at all receivers and the reservation of dedicated slots in the underlying TDMA scheme are the two key elements for temporal partitioning of VNs.

In case of event ports, *separate input ports* and thus separate queues ensure that the queuing delays for messages received from one sender job do not depend on the communication activities of other jobs (see Figure 4). In addition, separate message queues prevent a sender job that violates its message interarrival time specification [Kle75] from causing the loss of messages sent by other jobs. A message omission failure caused by a queue overflow at an input port only affects the messages sent by a single sender job.

In addition to providing separate input ports, we also need to prevent interference between messages on different communication channels prior to the arrival at the respective input ports. For this purpose, the DECOS architecture performs a separation of communication channels via statically reserved slots in the underlying TDMA scheme (cf. Section 2). Thereby, guardians can protect the access to these slots based on the a priori knowledge concerning the delimiting points in time of TDMA slots and the associations between TDMA slots and the structural elements of the integrated system. At all four levels identified in the hierarchic subdivision of TDMA slots (i.e., nodes, criticality domains, DASs, and jobs), protection of the respective slot is enabled. While several solutions for the protection of node slots on a time-triggered network are already available (e.g., [FPAF02, BKS03, Gua05]), middleware services for the protection of the subslots for criticality domains, DASs, and jobs have been realized in the scope of this work.

**Encapsulation at node-level by protecting node slots.** According to the DECOS fault hypothesis [OP06], a node is the fault-containment region for hardware faults. Since the justification for building ultra-reliable systems from replicated nodes rests on the assumption of failure independence among redundant units, the independence of fault-containment regions is of critical importance [BCV91]. Although a fault-containment region can restrict the immediate impact of a fault, error containment must ensure that errors cannot propagate across the boundaries of fault-containment regions. For hardware faults, an important goal of error containment is the protection of the node slots at the shared time-triggered network, i.e., preventing a faulty node from sending in the slot of another node.

**Encapsulation of criticality domains by protecting criticality-domain slots.** The DECOS architecture supports mixed criticality integration by sharing the nodes and the network among jobs of different criticality levels. In general, safety-critical and non safety-critical DASs will exhibit significant differences concerning the residue of design faults after deployment due to different development processes driven by economic constraints. In safety-critical DASs with reliability requirements of  $10^{-7}$  failures/hour or better (i.e., SIL 3/SIL 4 according to IEC 61508 [IEC99]), the absence of design faults can no longer be shown by testing alone. The achievement of reliability includes a rigorous development process, formal verification, and involvement of a certification agency. For this purpose correctness-by-construction methods have also gained more and more momentum in recent years for the development of safety-critical software [Dio04]. For non safety-critical DASs, certainty about the complete absence of design faults is usually economically infeasible. For this reason, the integrated architecture needs to prevent error propagation between different criticality domains to prevent a design fault in a non safety-critical DASs from affecting safety-critical ones.

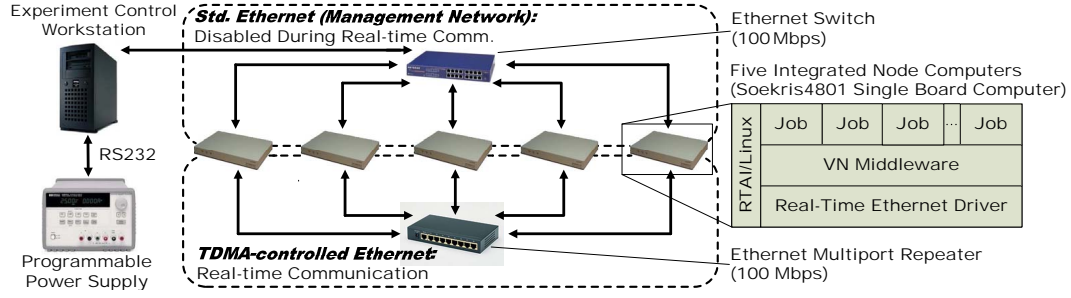


Figure 5. Implementation of Virtual Networks and Experimental Setup

**Encapsulation of DASs by protecting DAS slots.** Even within a particular criticality domain, encapsulation of DASs is beneficial in order to simplify the system integration and to improve robustness. If DASs are developed by different organizations (e.g., different suppliers), then encapsulation eases the task of tracing back a fault to the responsible organization.

**Encapsulation of jobs by protecting job slots.** Encapsulation at the job-level further improves the system integration and robustness benefits that can be gained by encapsulation at the DAS-level. In particular, if individual jobs are within the responsibility of different organizations or development teams, system integration and diagnosis efforts are reduced.

## 5. Experimental Setup with Prototype Implementation of Virtual Networks

VNs have been realized in a prototype implementation that serves as an experimental setup for the evaluation of the temporal performance and encapsulation properties. The experimental setup depicted in Figure 5 contains a distributed computer system with five nodes that are interconnected by a TDMA-controlled Ethernet network. The TDMA-controlled Ethernet network handles the message exchanges between the nodes hosting jobs of one or more DASs. Each DAS is provided with guaranteed communication resources via a dedicated VN realized on top of the TDMA-controlled Ethernet network.

### Nodes

Each node is shared among a number of jobs in order to overcome the prevalent “1 Function – 1 Electronic Control Unit” limitation of present-day electronic systems [BS05]. The so-called *VN middleware* realizes multiple VNs on top of the TDMA-controlled Ethernet network and provides a set of ports for the exchange of state or event messages to each of the jobs on the node.

**Hardware and Operating System.** Each node is implemented on a Soekris net4801<sup>2</sup> embedded single-board computer. A real-time Linux variant extended with a time-triggered task scheduler [HPOS05] is used in the construction of encapsulated partitions for the execution of the jobs. The operating system provides execution slots with a duration of 400  $\mu$ s for the execution of jobs and the virtual network middleware. Also, it monitors task execution times in order to detect worst-case execution time violations.

<sup>2</sup><http://www.soekris.com>

**Real-Time Ethernet Driver.** Time-triggered communication on top of Ethernet is a solution capable of ensuring predictable real-time behavior, while employing Commercial-Off-The-Shelf (COTS) hardware [KAGS05]. For the implementation of VNs, we have employed a software-based implementation of a TDMA-controlled Ethernet network. In each node a real-time Ethernet driver performs clock synchronization and periodic time-triggered transmissions and receptions of state messages. The real-time Ethernet driver provides access to these state messages via a set of state ports at the interface to the VN middleware.

The state ports include one output port for sending a state message on the TDMA-controlled Ethernet network and four input ports for receiving state messages from the other four nodes of the prototype cluster. We have used a homogeneous configuration with a single state message with a size of 1500 bytes for each node in the distributed system. The receptions and transmissions on the TDMA-controlled Ethernet network are strictly time-triggered, i.e., controlled by the progression of the global time.

**VN Middleware.** The VN middleware is a Linux kernel module that translates between the ports of the jobs and the ports towards the TDMA-controlled Ethernet network. The ports of the jobs include input and output ports, each storing either a state message (in case of a state port) or event messages (in case of an event port) as introduced in Section 4.

The ports towards the TDMA-controlled Ethernet network contain state messages with a compound structure, which represents the entities of the logical system view: criticality domains, DASs, and jobs. This compound structure of the state messages results from the hierarchic subdivision of the TDMA slots. At the finest granularity (i.e., at job-level), each data segment is uniquely associated with a specific sender job. Prior to a message transmissions on the TDMA-controlled Ethernet network, the VN middleware goes through all output ports at the interface to the jobs and constructs the job-specific data segments in the Ethernet message to be broadcast.

In addition, the VN middleware is activated after the (time-triggered) receptions of messages by the real-time Ethernet driver. In this case, the VN middleware processes all job-specific data segments in the Ethernet message and forwards the information to the input ports of the jobs.

The design decision of realizing the VNs in middleware on top of a time-triggered communication controller is motivated by the compatibility to existing time-triggered communication protocols and the preservation of existing validation results:

- **Compatibility with different time-triggered communication protocols.** Any time-triggered communication protocol, which provides a global time base and periodic exchanges of state messages, can be used as a basis for the realization of VNs. Examples for suitable protocols are TTP [KG94], SAFEbus [HD93], Time-Triggered Ethernet [KAGS05], and FlexRay [Fle05].
- **Preservation of validation results.** Since the time-triggered communication controller remains unchanged, the evidence for the correctness of the core algorithms of time-triggered communication protocols (e.g., for clock synchronization) is preserved.

**Application Software – Probe Jobs.** The jobs in the experimental setup are named *probe jobs*, because they stress the communication system in a controlled manner with predefined message send patterns and observe the resulting message receptions. The transmission of messages follows a *message send pattern*, which denotes for each output port and each round, the number and size of the messages that have to be transmitted. A probe job includes in each sent message a global timestamp of the send instant and a sequence number.

In addition, a probe job stores significant parameters of received messages (e.g., send and receive instant w.r.t. the global time base, sequence number), thus enabling an off-line analysis and an evaluation of the temporal properties of the VNs under different message patterns. A probe job polls upon each invocation all input ports for received messages. For each message read from an input port, the probe jobs constructs a *measurement record*:

⟨send instant, msg. sequ. number, receive instant, job id.⟩

The timestamp with the send instant and the message sequence number are part of the message content. The timestamp of the receive instant is determined by the probe job upon the reception of the message. The job identification denotes at which input port the message has been acquired. The job identification is expressed using the architecture-level namespace ( $n:s.v.j$ ) and denotes the sender job along with the respective VN and node hosting the sender job.

## Experiment Control Workstation and Management Network

The *experiment control workstation* (PC with Linux as its operating system) controls the experiments and imposes the following experimental sequence:

- *Reset of cluster.* The workstation resets the cluster every 90 sec. using a programmable power supply (HP E3631A) in order to start with defined initial state in each test run.
- *Generation of message patterns.* At the beginning of each test run, the workstation runs a *test pattern generator* that constructs control structures containing test patterns for the probe jobs. This tool uses the index of the test run and the identification of the campaign as parameters in order to create test patterns with a specific bandwidth consumption. The message model behind the test patterns is described in Section 8.
- *Transfer of message patterns.* The generated test patterns are transferred via Network File System (NFS) to the nodes. This transfer occurs via the *management network*, which is a standard Ethernet network (i.e., non real-time). On each node, the probe jobs load the test patterns of the current test run during startup.
- *Acquisition and storage of measurement records.* The probe jobs store the acquired measurement records in a file located in a directory of the workstation mounted via NFS. This back-transfer of the collected measurement records to the experiment control workstation uses the management network.

## 6. Hypotheses and Experiments

The experiments serve the purpose of evaluating the encapsulation of VNs in the temporal domain and the meeting of performance requirements for the targeted application-domain. These properties are key elements in order to enable the shift from federated to integrated architectures, while preserving fault isolation, complexity management, and system integration benefits of a federated solution in the integrated architecture.

Encapsulation is investigated w.r.t. different behaviors of jobs. We determine whether the application software comprising a job can (via its message transmissions) affect the temporal properties of messages exchanged by other jobs. In a first step, we are interested in the temporal effects under correct job behaviors. In general, the transmission behavior of a job transmitting event messages will adhere to an envelope that constrains the minimum and maximum interarrival times of messages. This envelope also constrains the job's load on the communication system.

Network Class	Examples of Protocols	Bandwidth	Typical Latencies	Application Domains
Class A	LIN	< 10 kbps	10-100ms	sensor/actuator
Class B	CAN	10kbps-125kbps	10-100ms	comfort domain
Class C	CAN	125kbps-1Mbps	5ms	powertrain domain
Class D	Byteflight	> 1 Mbps	5ms	multimedia, X-by-wire

Figure 6. SAE Network Classes

In a second step, we look at the temporal effects of faulty job behaviors, e.g., a job that transmits messages with a total bandwidth consumption exceeding the specified bandwidth. According to the DECOS fault hypothesis [OP06], a job forms a fault-containment region w.r.t. software faults. Thus, a particular software fault is restricted in its immediate impact to a single job. In the context of mixed-criticality systems, which are a major target domain of the DECOS architecture, it is vital to avoid error propagation via the shared communication system to other jobs.

We focus on the encapsulation of message transmissions in the VN middleware, which manages the communication resources provided by the underlying time-triggered communication network. The evaluation of the encapsulation for hardware faults, such as a node failure or a failure of the underlying time-triggered network (e.g., due to EMI) is not part of the experiments, because the handling of hardware faults through time-triggered communication protocols has already been evaluated extensively in previous work [ASBT03]. The experiments performed in this work are orthogonal to these hardware fault injection activities. In conjunction with such a dependable time-triggered network (e.g., TTP [KG94], FlexRay [Fle05], fault-tolerant Time-Triggered Ethernet [KAGS05]), the presented VNs also provide resilience against hardware faults.

The analysis of the performance of the VNs aims at quantifying the effect of temporal partitioning on the bandwidth and latencies of exchanged messages. As part of the experiments, we want to answer the question whether rigid temporal partitioning is achievable, while meeting the performance requirements imposed by present-day automotive applications and those envisioned for the future (e.g., X-by-wire).

## Hypotheses

The following two hypotheses have been evaluated in the experiments:

**Hypothesis 1 (Temporal Partitioning).** A VN provides predefined temporal properties (bandwidth, latencies). Hypothesis 1 consists of the following two subclaims:

SUBCLAIM 1.1 (TEMPORAL PARTITIONING BETWEEN DASs) *The message transmissions of the jobs in the VN of one DAS do not affect the temporal properties (bandwidth, latencies, variability of latencies) of messages exchanged in VNs of other DASs.*

SUBCLAIM 1.2 (TEMPORAL PARTITIONING WITHIN THE DAS) *The temporal properties (bandwidth, latencies, variability of latencies) of messages transmitted by a job are independent from the message transmissions of other jobs in the same DAS.*

**Hypothesis 2 (Performance).** Multiple VNs can be established on top of a time-triggered physical network to meet the performance requirements (i.e., bandwidth, latencies) of present day automotive networks.

Based on the performance four classes of in-vehicle networks can be distinguished (see Figure 6) according to the Society of Automotive Engineers (SAE) [LH02]. In present-day lux-

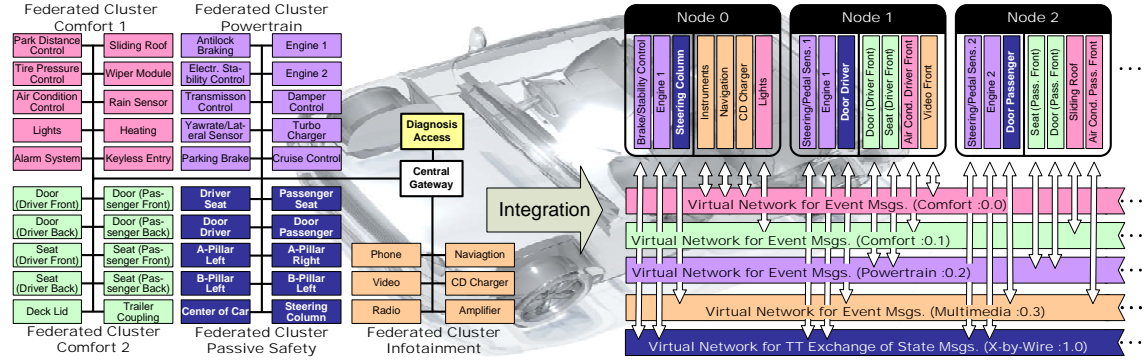


Figure 7. Mapping Physical Networks to Virtual Networks in the Integrated Architecture

ury cars, networks belonging to all four classes can be found. For example, in the BMW 7 series [Dei02] two class B networks (peripheral CAN and body CAN) interconnect the nodes of the comfort domain. A class C network (powertrain CAN with 500 kbps) serves as the communication infrastructure of the powertrain domain. In addition, the BMW 7 series is equipped with class D networks for multimedia (Media Oriented Systems Transport [MOS02]) and safety functions (Byteflight [BG00]). LIN fieldbus networks [ABD<sup>+</sup>99] for accessing low-cost sensors/actuators belong to SAE class A. Similarly, the communication architecture of the Volkswagen Phaeton comprises LIN fieldbus networks, two class B CAN networks for the comfort domain, a class C CAN network for drivetrain, and a class D network for multimedia [Leo04].

Hypothesis 2 consists of the following two subclaims, which are specified w.r.t. to the four network classes in Figure 6:

**SUBCLAIM 2.1 (MINIMUM BANDWIDTH)** *This subclaim states that the VNs on top of the underlying time-triggered physical network can provide the bandwidth of two class B networks (e.g., for comfort domain), one class C network (e.g., for powertrain domain), and two class D networks (e.g., for multimedia and X-by-wire). Class A networks are not addressed in the claim, because low-cost fieldbus networks (e.g., LIN) are assumed to remain as separate physical networks despite the shift to an integrated architecture [POT<sup>+</sup>05].*

**SUBCLAIM 2.2 (MAXIMUM LATENCIES)** *The maximum latencies of the VNs must be 10 ms to 100 ms in the body domain (class B networks) and in the order of ms in the chassis domain (class C networks) [Leo04]. For safety functions realized with class D networks, a reaction time of 5 ms is required [Dei02].*

## Virtual Network Configuration

The VN configuration, which has been used in the experimental evaluation, is based on the federated communication architecture of a typical present day automotive system. The VN configuration, which is depicted in Figure 7, enables the transition from a federated to an integrated architecture with multiple VNs. Each VN serves as a substitute for a respective physical network of a federated system. The VNs belong to five DASs, which are named according to the introduced architecture-level namespace with criticality domain 0 (non safety-critical) and criticality domain 1 (safety-critical).

For meeting the bandwidth requirements identified in hypothesis 2, the VN configuration supports two class B networks, one class C network, and two class D networks (see Figure 8). Two VNs with a bandwidth of 117 kbps (named comfort :0.0 and comfort :0.1 after the respective DASSs) support on-demand event message exchanges, e.g., like the medium-speed CAN networks deployed in the comfort subsystem of a car. A VN with a bandwidth of 492 kbps (named powertrain :0.2) provides the communication infrastructure of the powertrain subsystem of a car, thus replacing a high-speed CAN network.

Since CRC checks are handled by the underlying time-triggered communication protocol, the net bandwidths of the VNs exceed the net bandwidths of physical CAN networks with a raw bandwidth of 125 kbps or 500 kbps respectively. A further difference is that in a physical CAN network, the total bandwidth is potentially available to every node, but must be shared between them. The VN, on the other hand, assigns TDMA slots with their corresponding bandwidth exclusively to the jobs. In order to provide to each individual job the same maximum bandwidth in the prototype implementation, the entire VN is allocated the ten-fold bandwidth (1170 kbps or 4920 kbps) in case of the 10 jobs in the VN configuration. Consequently, the 117 kbps or 492 kbps are simultaneously available to all jobs. This allocation is possible, because the underlying physical Ethernet network provides a higher raw bandwidth (i.e., 100 Mbps) compared to the physical CAN networks. Due to bit arbitration, a physical CAN network is limited to 1 Mbps at a length of 40m [Bos91].

However, if the bandwidth is never simultaneously required by all jobs, sharing of the bandwidth between the jobs as in a physical CAN network would be more efficient. From this point of view, the higher bandwidth usage on the physical Ethernet network represents an overhead. In order to reduce this overhead, a priori knowledge concerning the communication behavior of the jobs can be used to assign to each job only the required fraction of the overall bandwidth. This means that the size of the job slots in the TDMA scheme would be reduced.

In addition to the VNs that serve as replacements for physical CAN networks, the configuration also contains a VN (multimedia :0.3) for the multimedia domain with a bandwidth of 492 kbps or 1496 kbps (depending on the job). A non-uniform bandwidth allocation is chosen, since some jobs may only transmit audio information, while other jobs also transmit audio and video information. Finally, the configuration includes a VN (X-by-wire :1.0) for the time-triggered exchange of state messages as required for safety-critical application subsystems.

Virtual Network	Node 0:	Node 1:	Node 2:	Node 3:	Node 4:
Comfort :0.0	:0.0.0 (117kbps)	:0.0.1 (117kbps)	:0.0.2 (117kbps)	:0.0.3 (117kbps)	:0.0.4 (117kbps)
	:0.0.5 (117kbps)	:0.0.6 (117kbps)	:0.0.7 (117kbps)	:0.0.8 (117kbps)	:0.0.9 (117kbps)
Comfort :0.1	:0.1.0 (117kbps)	:0.1.1 (117kbps)	:0.1.2 (117kbps)	:0.1.3 (117kbps)	:0.1.4 (117kbps)
	:0.1.5 (117kbps)	:0.1.6 (117kbps)	:0.1.7 (117kbps)	:0.1.8 (117kbps)	:0.1.9 (117kbps)
Powertrain :0.2	:0.2.0 (492kbps)	:0.2.1 (492kbps)	:0.2.2 (492kbps)	:0.2.3 (492kbps)	:0.2.4 (492kbps)
	:0.2.5 (492kbps)	:0.2.6 (492kbps)	:0.2.7 (492kbps)	:0.2.8 (492kbps)	:0.2.9 (492kbps)
Multimedia :0.3	:0.3.0 (1496kbps)	:0.3.1 (1496kbps)	:0.3.2 (492kbps)	:0.3.3 (492kbps)	:0.3.4 (492kbps)
X-by-Wire :1.0	:1.0.0 (617kbps)	:1.0.0 (617kbps)	:1.0.0 (617kbps)	:1.0.0 (617kbps)	:1.0.0 (617kbps)
	:1.0.5 (617kbps)	:1.0.5 (617kbps)	:1.0.5 (3117kbps)	:1.0.5 (3117kbps)	:1.0.5 (3117kbps)

Figure 8. Virtual Network Configuration



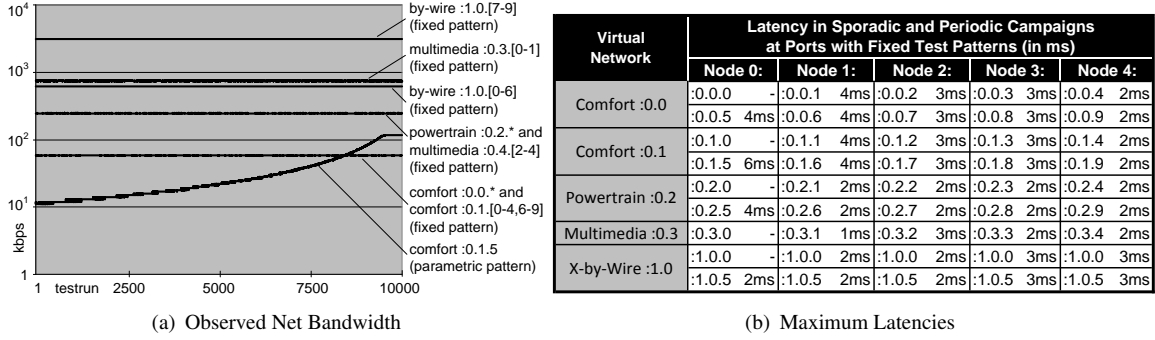


Figure 9. Observed Net Bandwidth and Maximum Latencies in Sporadic Campaign

## Experiments

Test patterns define the transmission behavior of the probe jobs w.r.t. a corresponding message model. In the following, test patterns for sporadic and periodic messages will be explained.

**Test Patterns.** Test patterns control the transmission requests issued by the probe jobs as described in Section 5. Each port is provided with a test pattern that controls the message transmissions performed by the probe job possessing the port. We distinguish between parametric and fixed test patterns:

- *Fixed test pattern.* The fixed test pattern exhibits the same predefined bandwidth utilization in all test runs.
- *Parametric test pattern.* A parametric test pattern exhibits a different bandwidth utilization in each test run. Within any specific test run, the bandwidth utilization in the parametric test pattern is constant. From the point of view of the entire experimental evaluation (i.e., all test runs), a parametric test pattern employs a variable bandwidth utilization, which is constrained by upper and lower bandwidth limits. The index of the current test run determines the bandwidth utilization. For example, starting with a lower bandwidth in the first test run, the bandwidth utilization is increased until reaching the upper limit for the bandwidth consumption in the last test run.

In the experiments, we have assigned a parametric test pattern to job :0.1.5, i.e., one of the jobs of DAS :0.1 (comfort). All other jobs have been assigned fixed test patterns. Job :0.1.5 with the parametric test pattern allows us to explore the effects of message transmissions with varying message interarrival times on the temporal behavior of VNs, both at the same port (i.e., the port with the parametric test pattern) and at other ports (i.e., ports with fixed test patterns).

**Message Timing Model for Test Patterns.** In the experiments, each job is assigned one or more messages that are repeatedly sent by the job. A job requests the dissemination of a message  $m$  at points in time  $\rho_{k,m} \in \mathbb{R}^+$  ( $k \in \mathbb{N}$ ), where  $\rho_{k,m}$  are stochastic variables. The parameter  $m$  identifies the message as well as the corresponding job, because each message is sent exclusively by a single job. The parameter  $k$  counts the instances of the message at that job. Every message  $m$  is characterized by two parameters, a minimum interarrival time  $d_m \in \mathbb{R}^+$  and a random interval offset  $\delta_{k,m} \in \mathbb{R}^+$ .

$$\forall k \in \mathbb{N} : \quad \rho_{k+1,m} - \rho_{k,m} = d_m + \delta_{k,m}$$

$d_m$  specifies an a priori known minimum interval of time between two transmission requests of  $m$ . The stochastic variables  $\delta_{k,m}$  cover the random part in the time interval between two transmission requests of  $m$ . For a particular message  $m$ , we assume that all stochastic variables  $\delta_{k,m}$  possess a uniform distribution  $U(0, u_m)$ .

This message model allows different message types to be distinguished. For a *sporadic message* the transmission request instants are not known, but it is known that a minimum time interval exists between successive transmission requests ( $\forall k \delta_{k,m} \sim U(0, u_m), d_m \in \mathbb{R}^+$ ). A *periodic message* has a constant time interval between successive message transmission requests ( $\forall k \delta_{k,m} = 0, d_m \in \mathbb{R}^+$ ).

**Campaigns.** The experiments consist of two campaigns each comprising 10,000 test runs. Campaign I employs a test pattern consisting of sporadic messages with varying minimum inter-arrival times, while campaign II explores the effects of different frequencies in periodic message exchanges. The reason for using two distinct campaigns is the evaluation of the effectiveness of the partitioning mechanisms for different types of messages (i.e., sporadic messages with random behavior and periodic messages with high predictability). Each of the two campaigns covers both correct and faulty job behaviors. The correct job behavior comprises message transmissions with a bandwidth consumption below the specified bandwidth limit of the VN (cf. Figure 8). An incorrect job behavior occurs when a job transmits more messages than permitted by the specified bandwidth constraints. The latter case represents a design fault, which is either a message timing failure of the job or a misconfiguration of the VN.

**Campaign I – Sporadic Messages.** Campaign I of the experiments comprises sporadic messages. One test pattern is parametric, while all other test patterns are fixed. In the fixed test pattern,  $d_m$  and  $u_m$  are selected for a 50% utilization of the bandwidth assigned to the port. In the parametric test pattern,  $d_m$  varies between  $166 \mu s$  and  $3.33 ms$ , while  $u_m$  varies between  $333 \mu s$  and  $6.66 ms$ . 10,000 sporadic messages perform an equidistant division of both parameter ranges ( $0 \leq n < 10000$ ):

$$\begin{aligned} d_m &= 3330 \mu s - (n/9999) \cdot (3330 \mu s - 166 \mu s) \\ u_m &= 6660 \mu s - (n/9999) \cdot (6660 \mu s - 333 \mu s) \end{aligned}$$

**Campaign II – Periodic Messages.** Periodic messages are used in campaign II of the experiments. In analogy to the previous campaign, all but one test pattern are fixed with  $d_m$  selected for a 50% utilization of the bandwidth assigned to the port. The parametric test pattern varies the fixed delay between  $333 \mu s$  and  $6.66 ms$ , while the random delay always equals 0:

$$d_m = 6660 \mu s - (n/9999) \cdot (6660 \mu s - 333 \mu s), \quad 0 \leq n < 10^4$$

## 7. Results

This section describes the results of the experiments. The observed bandwidths and transmission latencies for the parametric and fixed test patterns are presented. In addition, information concerning message omissions in the experiments is provided.

### Bandwidth

Figure 9(a) depicts the observed net bandwidth in the test runs of campaign I. Along the horizontal axis of the diagram, the test runs are distinguished. Along the vertical axes, the net band-

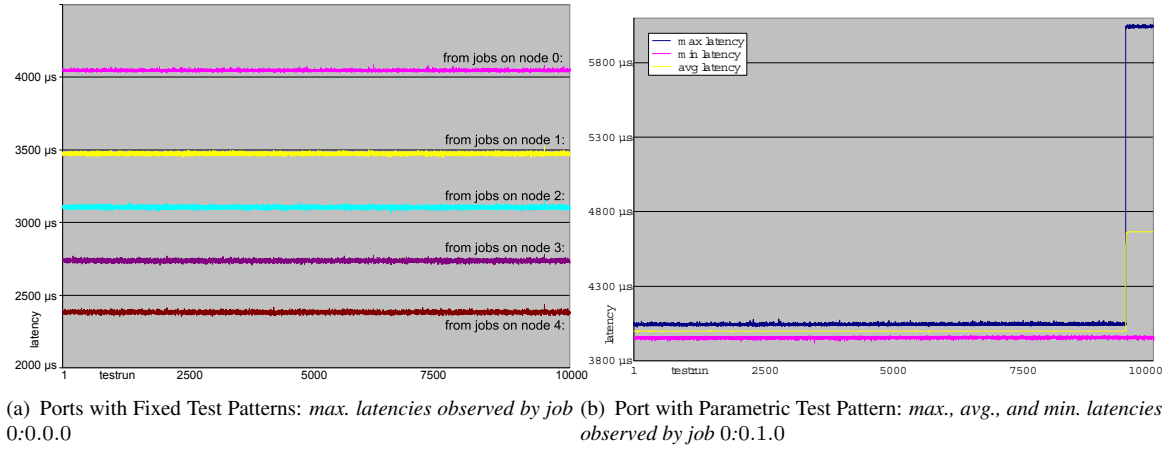


Figure 10. Latencies (y-axis denotes latency, x-axis denotes test run)

width of a test run is depicted (averaged over the duration of the complete test run). For all ports with fixed test patterns, the random interarrival times of the sporadic messages result in a variability ( $< 5\%$ ) of the observed net bandwidth in different test runs. However, this variability results only from the local differences in the random intervals between the sporadic message transmissions. As shown in Figure 9(a), the bandwidth variability does not change as the bandwidth consumption of job :0.1.5 increases. The initial bandwidth consumption of job :0.1.5 in test run 1 is 11.7 kbps. In test run 10,000, job :0.1.5 would require 234.6 kbps, but is confined to the pre-configured bandwidth of 117 kbps.

For the second campaign with periodic test patterns, the observed net bandwidths have exhibited the same characteristic progression as for the sporadic test patterns. However, in the second campaign no bandwidth variability has occurred at the ports with the fixed test patterns due to the absence of random interarrival times.

## Latencies

Figure 9(b) gives an overview about the observed maximum (end-to-end) latencies of the messages exchanged at each port with a fixed test pattern. The maximum latencies have been computed over all 10,000 test runs and were identical for campaigns I and II. The major reason for the identical maximum latencies in both campaigns is that except for the parametric test patterns, the messages produced by a job could always be transmitted using the subsequent job slot. Also, the periods of the periodic messages were not multiples or factors of the TDMA round length, thus leading to variable delays (like for the sporadic messages) between a job's message transmission requests and the dissemination through the underlying time-triggered network.

Important parameters resulting in these message transmission latencies on the VNs are the time-triggered activation times of the probe jobs in conjunction with the statically predefined slots for the exchange of messages on the underlying time-triggered network. The communication schedule employed in the prototype implementation consists of five slots for the five nodes. Each probe job is scheduled in a communication slot, i.e., synchronized with the communication schedule. Therefore, we can use a slot index (0...4) from the beginning of the round start to capture the following three parameters. The latencies depend on the phase shift relationship between the slot

	Slot 0	Slot 1	Slot 2	Slot 3	Slot 4		Slot 0	Slot 1	Slot 2	Slot 3	Slot 4		Slot 0	Slot 1	Slot 2	Slot 3	Slot 4
Node 0:	:0.0					Node 0:	:0.2					Node 0:			:0.3		
Node 1:		:0.0				Node 1:	:0.2					Node 1:					:0.3
Node 2:			:0.0			Node 2:	:0.2					Node 2:		:0.3			
Node 3:				:0.0		Node 3:	:0.2					Node 3:				:0.3	
Node 4:					:0.0	Node 4:					:0.2	Node 4:				:0.3	

Figure 11. Time-Triggered Scheduling of Probe Jobs

at which the sending probe job is scheduled ( $n_{pb}^s$ ), the slot at which the receiving probe job is scheduled ( $n_{pb}^r$ ), and the sender's slot  $n_{slot}^s$  in the TDMA scheme.

Based on these parameters, we can define two delays ( $d_{snd}$ ,  $d_{rcv}$ ) that contribute to the latency of a message exchanged on a VN. At the sending node, the *sender delay*  $d_{snd}$  results from the delay between the activation of the probe job and the start instant of the respective node's slot in the TDMA scheme. The receiver delay  $d_{rcv}$  is the delay between the start instant of the sending node's slot in the TDMA scheme and the activation of the receiving probe job. In the chosen VN configuration, each node sends exactly once in each TDMA round, thus a message transmission of a probe job is delayed until the following slot of the node that hosts the probe job. Formally, the sender and receiver delays are as follows:

$$d_{snd} = (1 + ((n_{slot}^s - n_{pb}^s - 1 + 5) \bmod 5)) \cdot d_{slot}$$

$$d_{rcv} = (1 + ((n_{pb}^r - n_{slot}^s - 1 + 5) \bmod 5)) \cdot d_{slot}$$

In the above formulas, the summand  $-1$  (inside the modulo expression) results from the need to sample the exchanged state messages at the end of the slot before the actual sending slot or the execution slot of the probe job. The summand 1 represents the delay for execution of the probe job.  $d_{slot}$  is the duration of a TDMA slot (i.e.,  $400 \mu s$  in the implementation).

In order to exemplify the sender and receiver delays in the implementation, Figure 11 depicts the execution slots for the probe jobs of selected DASs (:0.0, :0.2, and :0.3). For example, the transmission of messages on the VN of DAS :0.0 from node 0.1: to node 0.0: results in the parameters  $n_{pb}^s = 1$ ,  $n_{slot}^s = 1$ , and  $n_{pb}^r = 0$ . The resulting sum of the sender and receiver delay is 3.6 ms, which matches the value observed in the implementation. Likewise, the transmission of messages on the VN of DAS :0.2 from node 2: to node 0: results in the parameters  $n_{pb}^s = 2$ ,  $n_{slot}^s = 0$ , and  $n_{pb}^r = 0$ . In this case, the sum of the sender and receiver delay is 2 ms.

In addition, we have captured the progression of the latencies as the bandwidth consumption of the port with the parametric test pattern increases. Figure 10(a) depicts the latencies of one of the VNs in the 10,000 test runs for campaign I. As the bandwidth consumption of the port with the parametric test pattern at job :0.1.5 increases from 11.7 kbps to 234.6 kbps, no change in the maximum message transmission latencies at the ports with fixed test patterns is observable. The interval of observed message transmission latencies depends on the sender, with  $[4033 \mu s, 4086 \mu s]$  for node 0,  $[3449 \mu s, 3522 \mu s]$  for node 1,  $[3070 \mu s, 3151 \mu s]$  for node 1,  $[2706 \mu s, 2777 \mu s]$  for node 2, and  $[2355 \mu s, 2435 \mu s]$  for node 3. No correlation has been recognized between the bandwidth consumption of job :0.1.5 and the observed latencies at other jobs.

For the port with parametric test patterns, on the other hand, the bandwidth consumption beyond the specified bandwidth limit of 117 kbps in test runs 9,471 to 10,000 results in a variability of the latencies over the test runs. Maximum latencies of 6.1 ms have been observed by the jobs of DAS :0.1 in test runs with message transmissions exceeding the bandwidth of the VN (see Figure 10(b)). In these test runs, messages accumulate in message queues, thus requiring multiple

communication rounds for the exchange on the VN. The maximum latency is 4 ms before test run 9,471, when the bandwidth consumption is below 117 kbps.

## Message Omissions

For all ports with fixed test patterns no message omissions as indicated by the sequence numbers included in the messages have been observed. At the ports with parametric test patterns, message omissions were observed in case of transmission requests exceeding the pre-configured VN bandwidth (i.e., test runs 9,471 to 10,000). These test runs represent message timing failures, because job :0.1.5 does not comply to its specified bandwidth limit. No message omissions have been observed for the parametric ports before test run 9,471, i.e., when the bandwidth consumption is below the pre-configured 117 kbps.

## 8. Discussion of Results

The measurement results demonstrate that the communication system for an integrated architecture can be built with rigid temporal partitioning, while also providing competitive temporal performance (bandwidth, latency). The measurements have indicated that varying message loads created by a job do not affect the temporal properties of messages exchanged by other jobs.

### Hypothesis 1 – Temporal Partitioning

The experiments have demonstrated that the maximum and average transmission latencies of the ports with the fixed test patterns are independent from the behavior at the ports controlled by parametric test patterns. Smaller periods of periodic messages or smaller minimum interarrival times in case of sporadic messages have not resulted in the observation of increased transmission latencies at other ports. Consequently, both subclaims of hypothesis 1 hold. Interference in the transmission latencies of different ports was neither observed within a DAS nor between DASs.

The foundation for the effective temporal partitioning is the design decision of statically subdividing the communication slots according to the physical and logical structuring of the DECOS system (cf. Section 3). Based on this static resource allocation, the time-triggered communication protocol can protect the communication resources at the node-level, while the VN middleware protects the communication resources at the level of VNs and jobs.

A drawback of this design decision is a decreased flexibility w.r.t. extensions and modifications of the communication system. For example, in order to support the addition of nodes, either free slots need to be reserved at design time or a new time-triggered schedule needs to be computed and programmed into the nodes. Consequently, system designers need to assess the reduction of flexibility against the advantages of the static resource allocation (e.g., temporal composability, better complexity management, error containment for consequences of software faults).

### Hypothesis 2 – Performance

The experiments have shown that VNs on top of an underlying time-triggered network provide sufficient performance to support the communication requirements of present day automotive applications (see Section 6), as well as future safety-critical time-triggered DASs.

This result is particularly interesting as it demonstrates that temporal partitioning by the complete temporal isolation of DASs and jobs does not preclude competitive performance. Of course, there is a fundamental tradeoff between encapsulation and resource efficiency. Communication protocols, such as CAN, support the global multiplexing of bandwidth between all senders. In

contrast to the proposed static resource allocation, global bandwidth multiplexing leads to a more efficient use of the overall bandwidth in case of large communication loads that dynamically vary between senders. However, bandwidth utilization limits (e.g., 50% [AG03]) are introduced in order to control latencies (probabilistically since the utilization determines only the average load). Furthermore, bit arbitration constrains the maximum bandwidth [Bos91]. VNs do not exhibit these constraints. Neither is the raw bit rate constrained by bit arbitration, nor do bandwidth utilization constraints apply in order to guarantee message latencies.

**Bandwidth.** The configuration in the experimental setup supports two VNs that are equivalent to low-speed CAN networks, a VN equivalent to a high-speed CAN network, a VN for multimedia, and a VN for time-triggered communication. For the VN equivalent to low-speed CAN, each job is provided with a net bandwidth of 117 kbps. Since CRC checks are handled by the underlying time-triggered communication protocol, the net bandwidth of the VN exceeds the net bandwidth of a physical CAN network with a raw bandwidth of 125 kbps. For the VN equivalent to a high-speed CAN network, each job is provided with a net bandwidth of 492 kbps. In analogy to the low-speed VN, the realization of CRCs by the underlying time-triggered communication protocol makes the net bandwidth superior to a physical CAN network with a raw bandwidth of 500 kbps. The net bandwidth of the VN for multimedia (1496 kbps or 492 kbps) enables the playback of video or audio information. In the time-triggered VN, a net bandwidth of 3117 kbps or 617 kbps is available for safety-critical application subsystems.

**Latencies.** Depending on the scheduling of jobs, maximum latencies between 1.5 ms and 4.1 ms have been observed in the test cases with the fixed test patterns. These latencies meet the requirements of typical CAN-based automotive applications (e.g., order of ms in chassis domain, 10 ms to 100 ms in body domain [Leo04]). In particular, these latencies have been unaffected by job :0.5.0 executing the parametric test patterns. The latencies for the fixed test patterns have remained invariant even in case of a timing message failure of job :0.5.0, i.e., a message load exceeding the configured bandwidth. As a consequence of this message timing failure, only the faulty job :0.5.0 itself has experienced increased latencies of 6.1 ms and message omissions.

For the time-triggered message exchanges, a maximum latency of 2.8 ms has been observed, which is suitable for the implementation of safety-critical X-by-wire applications. For example, see [WSSLC03] for a discussion of the maximum delays for ensuring the safety in steer-by-wire systems.

## 9. Conclusion

This paper has shown that a time-triggered physical network is an effective foundation for establishing multiple virtual networks, each tailored to a respective application subsystem via its control paradigm (event message vs. state messages) and its temporal properties (e.g., bandwidth). The experimental assessment has yielded evidence that the realized VNs exhibit predefined temporal properties for the messages transmitted by a job, independently from the transmission behavior of other jobs and other application subsystems. In particular, rigid temporal partitioning is achievable, while at the same time meeting the performance requirements imposed by present-day automotive applications and those envisioned for the future (e.g., X-by-wire).

These results are particularly important in the context of the increasing complexity of embedded systems. System architects become forced to follow divide-and-conquer strategies that permit a

reduction of the mental effort for developing and understanding a large system by partitioning the system into smaller subsystems that can be developed and analyzed in isolation.

The temporal encapsulation of the communication resources belonging to subsystems, such as DAs or jobs in the DECOS architecture, is a key requirement for the constructive integration of integrated computer systems. By ensuring guaranteed temporal properties (e.g., bandwidth, latencies) for the messages transmitted by each job, prior services cannot be invalidated by the behavior of newly integrated jobs at the communication system. This quality of an architecture, which is denoted as temporal composability, relates to the ease of building systems out of subsystems. A system, i.e., a composition of subsystems, is considered temporally composable if the temporal correctness is not invalidated by the integration provided that temporal correctness has been established at the subsystem level.

VNs on top of a time-triggered network support temporal composability by ensuring that temporal properties at the communication system are not invalidated upon system integration. Furthermore, in the context of upcoming time-triggered technology in the automotive domain, the availability of a time-triggered communication network with high bandwidth enables the elimination of some of the physical networks deployed in present day cars. The communication resources of a single time-triggered network can be shared among different DAs. In conjunction with nodes for the execution of application software from different DAs, this integration not only reduces the number of node computers, but also results in fewer connectors and wires.

For future work, additional experiments are suggested as part of the development path towards the exploitation of VNs in ultra-dependable systems such as a drive-by-wire car. The experiments presented in this paper have focused on a single probe job within a selected VN. Interesting scenarios for future experimental evaluations include test cases with multiple probe jobs, which can exhibit simultaneous timing failures and are located in different VNs. Thereby, additional experiments can further increase the confidence in the presented hypotheses w.r.t. fault isolation and performance.

## Acknowledgment

This work has been supported in part by the European IST project ARTIST2 under project No. IST-004527 and the European IST project DECOS under project No. IST-511764.

## Bibliography

This work has been supported in part by the European IST project DECOS (IST-511764) and the European IST project ARTIST2 (IST-004527)

## References

- [ABD<sup>+</sup>99] Audi AG, BMW AG, DaimlerChrysler AG, Motorola Inc., Volcano Communication Technologies AB, Volkswagen AG, and Volvo Car Corporation. LIN specification and LIN press announcement. *SAE World Congress Detroit*, 1999.
- [Aer91] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Specification 651: Design Guide for Integrated Modular Avionics*, November 1991.
- [AG03] A. Albert and W. Gerth. Evaluation and comparison of the real-time performance of CAN and TTCAN. In *Proc. of 9th Int. CAN Conference*, Munich, 2003.
- [ASBT03] A. Ademaj, H. Sivencrona, G. Bauer, and J. Torin. Evaluation of fault handling of the time-triggered architecture with bus and star topology. In *Proc. of the 2003 Int. Conference on Dependable Systems and Networks*, pages 123–132, June 2003.

- [BCV91] R.W. Butler, J.L. Caldwell, and B.L. Di Vito. Design strategy for a formally verified reliable computing platform. In *Proc. of the 6th Annual Conference on Systems Integrity, Software Safety and Process Security*, pages 125–133, June 1991.
- [BG00] J. Berwanger and M. Peller R. Griessbach. Byteflight a new protocol for safety critical applications. In *Proc. of the FISITA World Automotive Congress*, Seoul, 2000.
- [BKS03] G. Bauer, H. Kopetz, and W. Steiner. The central guardian approach to enforce fault isolation in a time-triggered system. In *Proc. of the 6th Int. Symposium on Autonomous Decentralized Systems (ISADS 2003)*, pages 37–44, 2003.
- [Bos91] Robert Bosch GmbH, Stuttgart, Germany. *CAN Specification, Version 2.0*, 1991.
- [Bro87] F.P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, April 1987.
- [BS05] B. Bouyssounouse and J. Sifakis, editors. *Embedded Systems Design*. Springer Verlag, 2005.
- [Cri91] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.
- [DEC06] DECOS. Project deliverable D2.2.3 – Virtual communication links and gateways. Technical report, 2006.
- [Dei02] A. Deicke. The electrical/electronic diagnostic concept of the new 7 series. In *Convergence Int. Congress & Exposition On Transportation Electronics*, Detroit, MI, USA, October 2002. SAE.
- [DeL99] R. DeLine. *Resolving Packaging Mismatch*. PhD thesis, Carnegie Mellon University, Pittsburgh, June 1999.
- [Dio04] B. Dion. Correct-by-construction methods for the development of safety-critical applications. In *SAE 2004 World Congress & Exhibition*, Detroit, MI, USA, March 2004. SAE.
- [ea00] J. Penix et al. Verification of time partitioning in the DEOS scheduler kernel. In *Proc. of the 22nd International Conference on Software Engineering*, pages 488–497, 2000.
- [ea04] H. Heinecke et al. AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures. In *Proc. of the Convergence Int. Congress & Exposition On Transportation Electronics*. SAE, October 2004. 2004-21-0042.
- [Fle05] FlexRay Consortium. *FlexRay Communications System Protocol Specification Version 2.1*, May 2005.
- [FPAF02] J. Ferreira, P. Pedreiras, L. Almeida, and J. Fonseca. Achieving fault tolerance in FTT-CAN. In *Proc. of the 4th IEEE Int. Workshop on Factory Communication Systems*, 2002.
- [Gua05] FlexRay Consortium. BMW AG, DaimlerChrysler AG, General Motors Corporation, Freescale GmbH, Philips GmbH, Robert Bosch GmbH, and Volkswagen AG. *Node-Local Bus Guardian Specification Version 2.0.9*, December 2005.
- [HD93] K. Hoyme and K. Driscoll. SAFEbus. *IEEE Aerospace and Electronic Systems Magazine*, 8:34–39, March 1993.
- [Hei03] H.D. Heitzer. Development of a fault-tolerant steer-by-wire steering system. *Auto Technology*, 4:56–60, April 2003.
- [HPOS05] B. Huber, P. Peti, R. Obermaisser, and C. El Salloum. Using RTAI/LXRT for partitioning in a prototype implementation of the DECOS architecture. In *Proc. of the Third Int. Workshop on Intelligent Solutions in Embedded Systems*, May 2005.
- [IEC99] IEC: Int. Electrotechnical Commission. *IEC 61508-7: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems – Part 7: Overview of Techniques and Measures*, 1999.
- [KAGS05] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The Time-Triggered Ethernet (TTE) design. *Proc. of 8th IEEE Int. Symposium on Object-oriented Real-time distributed Computing (ISORC)*, May 2005.
- [KG94] H. Kopetz and G. Grunsteidl. TTP – A protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, 1994.
- [Kle75] L. Kleinrock. *Queuing Systems Volume I: Theory*. John Wiley and Sons, New York, 1975.
- [KLY00] D. Kim, Y.-H. Lee, and M. Younis. SPIRIT –  $\mu$ Kernel for strongly partitioned real-time systems. In *Proc. of the 7th Int. Conference on Real-Time Computing Systems and Applications*, 2000.
- [KO02] H. Kopetz and R. Obermaisser. Temporal composability. *Computing & Control Engineering Journal*, 13:156–162, 2002.



- [Kop97] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.
- [Leo04] J. Leohold. Communication requirements for automotive systems. In *Keynote Automotive Communication – 5th IEEE Workshop on Factory Communication Systems*, Vienna, Austria, September 2004.
- [LH02] G. Leen and D. Heffernan. Expanding automotive electronic systems. *Computer*, 35(1):88–93, January 2002.
- [MOS02] MOST Cooperation, Karlsruhe, Germany. *MOST Specification Version 2.2*, November 2002.
- [OH06] R. Obermaisser and B. Huber. Model-based design of the communication system in an integrated architecture. In *Proc. of the 18th International Conference on Parallel and Distributed Computing and Systems*, pages 96–107, 2006.
- [OP05] R. Obermaisser and P. Peti. Specification and execution of gateways in integrated architectures. In *Proc. of the 10th IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA)*, Catania, Italy, September 2005. IEEE.
- [OP06] R. Obermaisser and P. Peti. A fault hypothesis for integrated architectures. In *Proc. of the 4th Int. Workshop on Intelligent Solutions in Embedded Systems*, June 2006.
- [OPK05] R. Obermaisser, P. Peti, and H. Kopetz. Virtual networks in an integrated time-triggered architecture. In *Proc. of the 10th IEEE Int. Workshop on Object-oriented Real-time Dependable Systems (WORDS2005)*, 2005.
- [Par01] *Software Fundamentals: Collected Papers by David L. Parnas*. Addison-Wesley, April 2001.
- [POT<sup>+</sup>05] P. Peti, R. Obermaisser, F. Tagliabo, A. Marino, and S. Cerchio. An integrated architecture for future car generations. In *Proc. of the 8th IEEE Int. Symposium on Object-oriented Real-time distributed Computing*, May 2005.
- [RTC92] Radio Technical Commission for Aeronautics, Inc. (RTCA), Washington, DC. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, December 1992.
- [Rus99] J. Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Also to be issued by the FAA.
- [Rus01] J. Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, SRI Intern., 2001.
- [Sif05] J. Sifakis. A framework for component-based construction. In *Proc. of 3rd IEEE Int. Conference on Software Engineering and Formal Methods (SEFM05)*, pages 293–300, September 2005.
- [Sim96] H.A. Simon. *The Sciences of the Artificial*. MIT Press, 1996.
- [SM99] J. Swingler and J.W. McBride. The degradation of road tested automotive connectors. In *Proc. of the 45th IEEE Holm Conference on Electrical Contacts*, pages 146–152, October 1999.
- [TBDP98] E. Totel, J. P. Blanquart, Y. Deswarte, and D. Powell. Supporting multiple levels of criticality. In *Proc. of the The 28th Annual International Symposium on Fault-Tolerant Computing*, page 70, Washington, DC, USA, 1998.
- [WSSLC03] C. Wilwert, Y. Song, F. Simonot-Lion, and T. Clement. Evaluating quality of service and behavioral reliability of steer-by-wire systems. In *Proc. of 9th IEEE Int. Conference on Emerging Technologies and Factory Automation*, 2003.



# REUSE OF CAN-BASED LEGACY APPLICATIONS IN TIME-TRIGGERED ARCHITECTURES

*IEEE Transactions on Industrial Informatics, Volume 2, Number 4, 2006, pages 255–268.*

Roman Obermaisser  
Vienna University of Technology  
Real-Time Systems Group  
romano@vmars.tuwien.ac.at

**Abstract** Upcoming car series will be deployed with time-triggered communication protocols due to benefits with respect to bandwidth, predictability, dependability, and system integration. In present day automotive networks, CAN is the most widely used communication protocol. Today, up to five CAN buses and several private CAN networks result from the bandwidth limits of CAN in conjunction with constraints concerning bus utilization aimed at controlling transmission latencies. In this context, the upcoming introduction of time-triggered networks into series production offers the potential to reduce the number of CAN networks by exploiting the high-bandwidth of the time-triggered network instead of CAN buses. Due to the elimination of CAN buses, the resulting reduction of wiring and connectors promises a significant reduction in hardware cost and reliability improvements. In order to support the reuse of existing CAN-based application software, this paper presents a solution for the emulation of a CAN communication service on top of an underlying time-triggered network. By providing to CAN-based applications the same interface as in a conventional CAN system, redevelopment efforts for CAN-based legacy software are minimized. For this purpose, a CAN emulation middleware operates between a time-triggered operating system and the CAN-based applications. In a first step, the middleware establishes event channels on top of the time-triggered communication network in order to support on-demand transmission requests at a priori unknown points in time. The middleware then emulates the CSMA/CA media access protocol of a physical CAN network for passing messages received via event channels to the application in the correct temporal order. Finally, the API of the widely-used HIS/VectorCAN driver provides a handle-based programming interface with support for message filtering and callbacks. A validation setup with a TTP cluster demonstrates that the CAN emulation can handle CAN-based legacy software and a real-world communication matrix provided by the automotive industry.

**Keywords:** Computer network performance, distributed algorithms, legacy systems, real-time systems, road vehicle electronics

## 1. Introduction

Time-Triggered (TT) and Event-Triggered (ET) control are two different paradigms for the construction of the communication service of a distributed real-time system [Kop97]. Communication protocols with ET control (e.g., Controller Area Network (CAN) [Bos91]) are prevalent in current automotive networks, because they offer high flexibility and resource efficiency. However, with the increasing criticality of automotive computer systems (e.g., advanced driver assistance systems [LS04], X-by-wire [Hei03]), there is an upcoming shift from ET to TT communication protocols [LH02]. Communication protocols with TT control (e.g., TTP [TTT02a], SafeBus [HD93],

FlexRay [Fle05]) excel with respect to predictability, composability, error detection and error containment. For this reason, in-vehicle electronic systems will be deployed with TT communication protocols beginning in 2007 [For04].

At present, CAN [Bos91] is the most widely used automotive protocol with a 100% market penetration. Major advantages of CAN include its high flexibility (e.g., migration transparency, no need to change a communication schedule when adding ECUs), resource efficiency through the sharing of bandwidth between ECUs, low cost of CAN hardware, and high availability of CAN-based tools (e.g., [Vec05]) and engineers with CAN know-how (e.g., experience in production process, existing maintenance chain, field experience).

In the context of upcoming TT technology in the automotive domain, the availability of a TT communication network with high bandwidth (e.g., 10 Mbps in FlexRay [Fle05]) enables the elimination of one or more of the physical CAN networks deployed in present day cars. The communication resources of a single TT network can be shared for the exchange of both TT messages and CAN messages. In conjunction with integrated ECUs, i.e., ECUs for both CAN-based application software and application software based on TT communication, the sharing of communication and computational resources not only reduces the number of ECUs, but also results in fewer connectors and wires. Fewer connectors and wires not only decrease hardware cost, but also lead to improved reliability. Field data from automotive environments has shown that more than 30% of electrical failures are attributed to connector problems [SM99].

Nevertheless, despite the shift to TT communication protocols, previous investments motivate the reuse of CAN-based legacy applications. In addition, this strategy allows to retain software with conceivably low field failure rates, i.e., for those applications that have functioned correctly in a large set of cars. In order to minimize redevelopment efforts of legacy software (e.g., in comfort domain, powertrain domain), when migrating existing CAN-based applications to the TT system, there is the need to reestablish the services of the platform, for which the legacy software has been developed for. Among these services of the legacy platform are the operating system services (e.g., OSEK/VDX [OSE05]) and the CAN communication service. The focus of this paper is the establishment of the CAN communication service, i.e., the application's interface to the underlying communication system. For this CAN communication service, three main requirements can be identified, which need to be satisfied for ensuring the correctness of CAN-based legacy applications after the migration to the TT computer system:

- 1 **Support for ET Control.** CAN legacy software can request message transmissions on-demand at a priori unknown instants. In order to use a common network for both TT messages and CAN message exchanges, both TT and ET control need to be supported by the communication system.
- 2 **Temporal Properties.** In general, the correct behavior of CAN legacy software depends on the temporal properties of a CAN communication system. Relevant properties include the bandwidth, the transmission latencies, and the temporal order of messages.
- 3 **API.** Legacy CAN software accesses the communication system through a specific API. The establishment of this API is thus a prerequisite for reuse with a minimum of redevelopment efforts.

This paper presents a solution for the emulation of a CAN network on top of an underlying TT communication protocol. This so-called *Virtual CAN Network (VCN)* can be established on top of different TT communication protocols (e.g., TTP, FlexRay). In contrast to other solutions for ET/TT integration (see Section 2), this solution addresses all three requirements identified above.

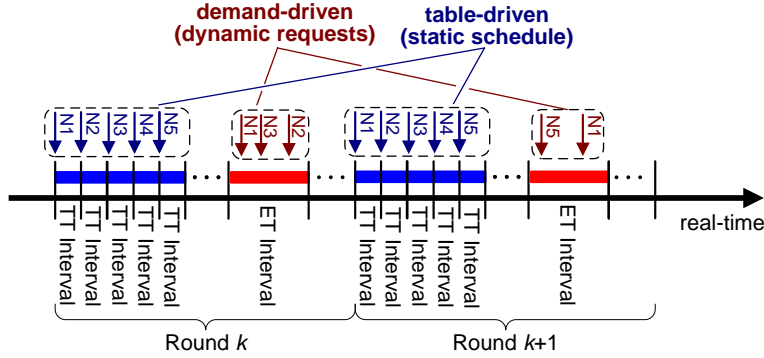


Figure 1. Communication Intervals for ET and TT Messages

An *event service* maps TT control to ET control by realizing event channels on top of an underlying TT communication protocol for the on-demand transmission of messages at a priori unknown points in time. Event channels support the temporal properties of a physical CAN network through a predefined relationship between the communication schedule of the TT physical network and the bandwidth and latencies of a VCN. In addition, an emulation (denoted as *protocol emulation service*) of the Carrier Sense Multiple Access Collision Avoidance (CSMA/CA) media access control protocol ensures that the temporal message order is identical to the one of a physical CAN network. Finally, the *front-end* is a service that maps the VCN onto the API expected by legacy applications (e.g., API of HIS/VectorCAN driver).

A prototype implementation using the Time-Triggered Protocol (TTP) [TTT02a] serves as a target for the validation of the devised solution. The validation activities have included measurements of the transmission latencies, bandwidth, and temporal message order of VCNs with a communication matrix from a series-car provided by an automotive manufacturer as inputs. These results have been compared to the transmission latencies, bandwidths, and temporal message orders of a physical CAN network, as indicated by a MATLAB/Simulink simulation framework of a physical CAN network.

The paper is structured as follows. Section 2 gives an overview of related work on the integration of ET/TT communication protocols. The system model of a TT computer system with support for VCNs is the focus of Section 3. The following sections are devoted to explaining the realization of the architectural services introduced in this system model. Section 4 describes the event service for transforming ET into TT communication services. The establishment of the correct temporal message order is the focus of Section 5. The front-end for establishing legacy CAN APIs is the topic of Section 6. In combination, these three services (event service, protocol emulation, front-end) cover the three requirements concerning a VCN identified above. Finally, a validation setup with a prototype implementation in Section 7 provides experimental evidence on the ability of the VCNs to support the reuse of CAN-based legacy applications.

## 2. Related Work of Integration of ET and TT Communication

The VCN presented in this paper provides an ET communication services for the dissemination of CAN messages via an overlay network on top of an underlying TT network. This section gives an overview of other solutions for the integration of ET and TT control, all of which perform the integration of the two control paradigms at the Media Access Control (MAC) layer. As depicted

in Figure 1, these protocols distinguish two types of reoccurring intervals: ET and TT intervals. The arrows represent the message transmission start instants with the sender node superscripted to each arrow. A TT interval permits a single node to send a message after the a priori specified start instant of the interval. The start and end instants of such a periodic TT message transmission, as well as the respective sending node computer are fixed at design time. For this class of messages, contention is resolved statically. All TT message transmissions follow an a priori defined schedule, which repeats itself after each communication round. In ET intervals, message exchanges depend on transmission requests from the application (i.e., external control) and the start instants of message transmissions can vary. Furthermore, ET intervals can be assigned to multiple (or all) node computers of the system. For this reason, the MAC layer needs to support the dynamic resolving of contention when more than one node computer intends to transmit a message. During ET intervals a sub-protocol (e.g., CSMA/CA, CSMA/CD) takes over that is not required during TT intervals in which contention is prevented by design.

While TT messages can always be scheduled to fit into the respective intervals at design time, on-demand ET message transmissions require support by the MAC protocol for protecting TT intervals. The mechanism for the delimitation of ET and TT intervals allows to further classify protocols for ET/TT integration at the MAC layer: contention avoidance protocols and contention tolerant protocols.

### ET/TT Contention Avoidance

Contention avoidance protocols reserve at the end of each ET interval a time interval, in which no message transmissions may be started. The length of this time interval is equal to the maximum message transmission duration of an ET message. Consequently, it is ensured that an ET message transmission can always be completed before the next TT interval starts.

An example for a communication protocol realizing this solution is FlexRay [Fle05]. In each communication round, FlexRay supports a single ET interval denoted as the *dynamic segment* and multiple TT intervals forming a continuous time interval named the *static segment*. The static segment realizes a strict TDMA scheme, while the dynamic segment employs an event-driven mini-slotting sub-protocol. The fixed duration of the dynamic segment is subdivided into mini-slots that identify potential start times of message transmissions. Each node computer counts the number of idle mini-slots and is assigned one or more unique counter values. If during a dynamic segment the bus has been idle for a number of mini-slots equal to one of these counter values, the respective node can send a message. During the transmission of a message, the incrementing of the idle mini-slot counter is paused. Consequently, a smaller counter value gives a node computer a higher priority compared to node computers with larger counter values. Due to the demand driven access pattern, the reserved bandwidth of a dynamic segment can be shared between node computers. In FlexRay, the time interval reserved for contention avoidance at the end of an ET interval is part of the dynamic slot idle phase.

Time-Triggered CAN (TTCAN) [TBW<sup>+</sup>00] and Flexible Time-Triggered CAN (FTT-CAN) [PA00] in controlled mode are further examples of protocols integrating ET and TT communication at the MAC Layer with contention avoidance. TTCAN and FTT-CAN are CAN-based master/slave protocols for building a TT communication service on top of CAN, while also permitting ET CAN communication. TT intervals (denoted as exclusive windows in TTCAN) support periodic TT messages, which are scheduled statically by an off-line tool in order to prevent collisions. ET intervals (denoted as arbitrating windows in TTCAN) resolve contention dynamically with the CSMA/CA arbitration mechanism of CAN. In TTCAN and the controlled mode of FTT-CAN,

a node computer may only send an ET message, if the remaining time interval before the next TT message has a sufficient length for preventing any interference of TT and ET messages.

## ET/TT Contention Tolerant Protocols

ET/TT contention tolerant protocols do neither restrict transmission start instants within an ET interval nor preempt ongoing ET message transmissions. All ET message transmissions are permitted to finish, thus leading to potential perturbations of the boundaries of TT intervals.

An example for this protocol type is FTT-CAN in *uncontrolled mode* [PA00]. In analogy to the controlled mode of FTT-CAN, both ET communication and TT communication are supported. Every node participating in TT communication is equipped with a local table that contains information about the TT messages transmitted and received during each communication round. Node computers can start with the transmission of ET messages at arbitrary instants. Through assigning higher priorities to TT messages, it is ensured that TT messages always win in the arbitration process. Nevertheless, the non-preemptive nature of CAN results in transmission jitter of TT messages, in case an ET message is being transmitted when a TT message transmission is scheduled. In the worst-case, a TT interval with one or more TT messages is delayed by the maximum transmission duration of an ET message.

## Relationship to Virtual Networks

A major reason for choosing an ET overlay network for the realization of the virtual CAN network is the ability to integrate this solution into different TT communication protocols (e.g., TTP [TTT02a], FlexRay [Fle05], SAFEbus [HD93]). Virtual CAN networks build on top of the periodic exchange of state messages, which is supported by all TT communication protocols, while refraining from the exploitation of the MAC layer ET communication service that is specific to each of the protocols.

Unlike the VCN presented in this paper, other ET/TT integration solutions either inherit the limitations of CAN or do not explicitly focus on the reuse of CAN-based legacy applications. By building on top of CAN, FTT-CAN naturally supports CAN-based legacy applications, but also inherits all limits with respect to bandwidth and dependability of the CAN protocol. For example, CSMA/CA requires bits to stabilize on the channel, i.e., the bit length must be at least as large the propagation delay. The resulting maximum bandwidth of 1 Mbps for a network of 40 m is insufficient, when integrating multiple CAN buses in combination with TT message exchanges on a single network. In uncontrolled mode, there is the additional communication jitter of TT messages induced by standard CAN messages, which results in an adverse effect on the quality of control in jitter-sensitive applications (e.g., control loops).

The ET communication service of FlexRay, on the one hand, does not establish the temporal message order of a physical CAN network through an emulation of the CSMA/CA media access control protocol. Secondly, no mapping to a CAN API, such as the widely used HIS/VectorCAN driver API, has been performed. Both issues are requirements for the reuse of CAN-based legacy applications without redevelopment efforts. Nevertheless, the TT communication service of FlexRay is a suitable as a baseline for the establishment of a VCN as an overlay network.

## 3. Controller Area Network Emulation

Physically, the distributed system for the integration of CAN and TT communication consists of a set of *integrated nodes* and a single *TT network* that interconnects these nodes. Each integrated

node consists of a host computer and a communication controller. The communication controller executes a TT communication protocol to provide a TT message transport service, a global time base, and a membership service. Any communication protocol that provides these services can be used as a basis [Rus01] for the integration of CAN and TT communication. Examples of suitable communication protocols are the Time-Triggered Protocol (TTP) [TTT02a] and FlexRay [Fle05], when extended with a membership service. The three services, which abstract from the implementation technology of the underlying TT communication protocol, are explained in the following:

- **TT message transport service.** This service performs periodic TT exchanges of state message. At each node the communication controller (e.g., TTP controller C2 [TTT02c], FlexRay Controller MFR4200 [Fre05]) provides a memory element with outgoing state messages that are written by the application and read by the communication controller prior to broadcasting them on the TT network. In addition, the memory element contains incoming state messages that are read by the application and updated by the communication controller with state messages read from the TT network (i.e., information broadcast by other nodes). This memory element, which is denoted CNI in TTP and Controller Host Interface (CHI) in FlexRay, is provided by most TT communication protocols with syntactic differences of state messages (e.g., header format) and protocol-specific constraints (e.g., only one message sent by a node per communication round in TTP [TTT02a], same size for all state messages in FlexRay [Fle05]).
- **Global time.** In a distributed computer system, nodes capture the progression of time with physical clocks containing a counter and an oscillation mechanism. An observer can record the current granule of the clock to establish the *timestamp* of an event. Since any two physical clocks will employ slightly different oscillators, the time-references generated by two clocks will drift apart. Clock synchronization is concerned with bringing the time of clocks in a distributed system into close relation with respect to each other. IEEE 1588 [IEE02] is an example for a protocol to synchronize nodes of a distributed system to a high degree of accuracy and precision. By performing clock synchronization in an ensemble of local clocks, each node can construct a local implementation of a global notion of time.

For the emulation of CAN, we require the ability to assign timestamps w.r.t. to a global time base to events, such as a transmission request of the application. These timestamps are the basis for ensuring the correct temporal order of CAN messages, which will be discussed in Section 5. For example, in TTP a global time base with a precision down to  $1\ \mu\text{s}$  (depending on clock drifts) is provided by a macrotick counter at the controller-host interface [TTT02a]. Similarly, FlexRay [Fle05] offers a *cycle counter* denoting the number of the current communication round (denoted as *cycle* in FlexRay) and a *macrotick counter* within the cycle.

- **Membership service.** The *membership service* provides consistent information about the operational state (correct or faulty) of nodes [Cri91]. In a TT communication system the periodic message send times are membership points of the sender [KGR91]. Every receiver knows a priori when a message of a sender is supposed to arrive, and interprets the arrival of the message as a life sign of the sender.

TTP, which has been used in the prototype implementation of the CAN emulation, natively provides a membership service. For other protocols, such as FlexRay, a membership service can be layered on top of the protocol services. For example, in [SKR97] a membership



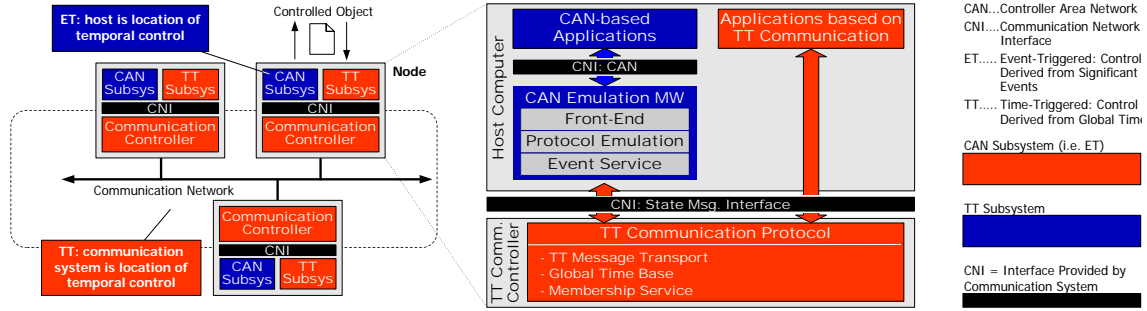


Figure 2. Integrated Node Computer with CAN Emulation Middleware

service is presented that is based on a generic model of a TT system that is not restricted to a specific communication protocol.

On top of the communication controller providing these three services, the host computer runs the application software modules, which are either based on TT communication or require a CAN communication service. For this purpose, the host computer of an integrated node (see Figure 2) contains two subsystems: an ET subsystem for CAN-based applications and a TT subsystem for applications based on the periodic exchange of state messages. In the latter execution environment, the application software directly exploits the services of the communication controller. The CAN execution environment employs a *CAN emulation middleware* comprising three layers (event service, protocol emulation, and front-end) for performing a step-wise transformation of the TT services into a CAN communication service.

The *event service* establishes *Event Channels (ECs)* for the on-demand transmission of messages. Message transmission requests can occur at arbitrary instants, but the dissemination of the messages on the underlying TT network is always performed at the predefined global points in time of the TDMA slots. The middleware service for *protocol emulation* exploits the ECs provided by the event service in order to realize a Virtual CAN Network (VCN). The protocol emulation establishes the temporal message order of a physical CAN network by performing at run-time a simulation of the CSMA/CA media access control strategy of a physical CAN network within each node of the TT system. The third middleware service establishes a CAN-based higher protocol and provides an *API* by which application software can access a VCN. The *front-end* provides the operations for the transmission and reception of messages and the reconfiguration of the CAN communication system (e.g., setting of message filters). The front-end also implements interrupt mechanisms via callbacks that enable the application to react to significant events, such as the reception of a message or the occurrence of a fault.

The following sections provide a detailed discussion on the event service, the protocol emulation, and the front-end.

#### 4. Event Service

The purpose of the *event service* is the mapping of the state message interface provided by the TT communication controller onto an event message interface that is the basis for all higher layers (i.e., the protocol emulation, the front-end, and the application). The event message interface comprises queues with support for ET on-demand transmission requests.

## State Message Interface

The state message interface of the underlying TT communication system is also known as a temporal firewall [KN97] due to the absence of control flow from the application to the communication system. The interface between the communication system and the application software are state messages that are read (in case of a message received by the node) or written (in case of a message sent by the node) by the communication system at a priori specified instants w.r.t. to a global time base. A temporal firewall is based on the principle of *updates in place*, i.e., idempotent state messages are overwritten when a more recent version of the state messages becomes available. Updates in place require each message to be self-contained, i.e., the data contained in a message is an absolute value that can be interpreted independently from previous versions of the message. This self-contained nature of messages is denoted as *state semantics*.

The application software is aware of the predefined instants, at which the communication system accesses the state messages. The application software uses this a priori knowledge to fulfill the sender and receiver obligations of a temporal firewall. The sender obligation consists in updating those state message that are sent by the respective node at a sufficiently low update period that ensures temporal accuracy of the state message at the receivers. Based on the a priori knowledge about the temporal accuracy of the real-time images in the temporal firewall, the consumer must sample the information in the temporal firewall with a sampling rate that ensures that the accessed real-time image is temporally accurate at its time of use.

## Event Message Interface

In contrast to the TT applications, CAN-based applications do not know upfront about the instants of messages exchanges at the communication system. A CAN communication system reacts dynamically to transmission requests, transmitting messages (normal CAN frames and request CAN frames [Bos91]) only in response to a transmission request from the application. The transmission requests from CAN applications are not synchronized to the underlying TT communication protocol. Furthermore, in general only an average message transmission frequency is known for ET applications. For example, in CAN-based automotive systems a bus utilization in the range of 50% is demanded for non safety-critical applications [AG03]. The bus utilization is a statement about the average frequency of message transmissions on the CAN bus, permitting the exceeding of the number of messages per second during limited intervals of time. The underlying TT communication system, however, allows to send exactly  $n$  data bytes during each communication round, where  $n$  (e.g., up to 254 in FlexRay [Fle05], up to 240 in TTP [TTT02a]) depends on the duration of the CAN subslot and the bitrate of the TT physical network. Hence, it can occur that within a communication round multiple transmission requests occur, while none occur in other communication rounds. In order to buffer messages that cannot be disseminated immediately, the interface of the event service to the higher layers are message queues. With messages queues, the event service also supports limited intervals of time during which the transmission of more messages is requested by the application than can be disseminated via the communication system.

## Mapping of State Message Interface to Event Message Interface

The media access protocol of the TT network is TDMA, which statically divides the channel capacity into a number of slots and assigns to each node a unique slot that periodically reoccurs at a priori specified global points in time. Support for CAN communication results from the

temporal subdivision of the communication resources provided by this TDMA scheme. Each node's slot is subdivided into two subslots, namely a slot for TT communication and a slot for the ET dissemination of CAN messages (see Figure 3).

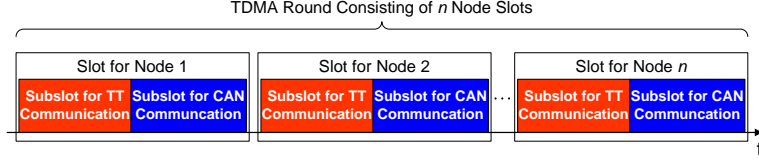


Figure 3. Temporal Subdivision of TDMA Slots

The event service disseminates the messages in the outgoing queues via the bandwidth provided via the CAN subslots. Although message transmission requests can occur at arbitrary instants, the dissemination of the messages on the underlying TT network is always performed at the predefined global points in time of the CAN subslots. The event services performs a sampling of the ET message transmission requests and defers their transmission until the next CAN subslot of the node occurs in the TDMA scheme. During each communication round, the event service transfers CAN messages to be sent from the outgoing message queues of the event message interface into the temporal firewall interface, namely into the state variables that are broadcast by TT communication controller. In addition, the event service retrieves received messages at the temporal firewall interface and forwards them into the incoming messages queues. Message fragmentation, i.e., the filling of CAN subslots with packets from as many messages as possible, ensures that the CAN subslots are optimally utilized in case of varying message sizes. In addition, message fragmentation decouples the duration of CAN subslots from the maximum message sizes. CAN subslots can be shorter than a CAN message (e.g., CAN subslot with a single byte), thus permitting short round lengths despite the support for CAN communication. Short round lengths are important for control applications that require high update frequencies of state variables.

The event service constructs for each node an *Event Channel (EC)*, via which the node can broadcast its event messages to all other nodes of the distributed system (i.e., point-to-multipoint topology). In order to support a general communication topology, in which each node can transmit CAN messages, a system with  $n$  nodes employs  $n$  ECs with an equal number of CAN subslots (see Figure 4).

An overview of the functionality of the event service for establishing one outgoing EC and multiple incoming ECs at a node is depicted in Figure 5. The part that is responsible for the transmission of event messages (starting at Line 8) is activated prior to the node's CAN subslot in the TDMA scheme of the TT communication protocol. The activation occurs when the current time  $t_{\text{now}}$  is equal to the start instant  $t_{\text{slot}}$  of the node's CAN subslot subtracted by the worst-case execution time  $d_{\text{mw}}$  of the CAN emulation middleware. This activation time ensures that the update of the outgoing state message is finished before the TT communication controller reads the state message at instant  $t_{\text{slot}}$ . For the fragmentation of event messages, the event service maintains a partially sent event message  $m_s$ . Upon each invocation, the event service extracts a packet from this partially sent event message and places the packet into the outgoing state message. Furthermore, the event service fetches additional event messages from the protocol emulation in case space for packets of more than one event message is available in the state message.

The reception part in Figure 5 starts at Line 16. The event service maintains a vector containing for each node  $i$  a partially assembled event message  $m_r[i]$ . Upon the (TT) arrival of a state

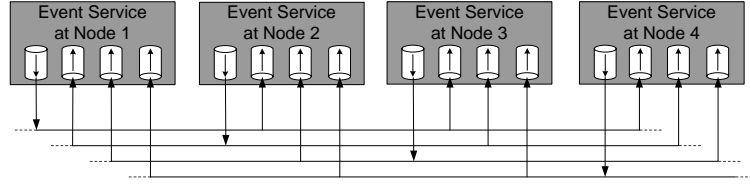


Figure 4. Event Channels on top of the Time-Triggered Physical Network

```

1 // Variable description
2 //  $m_r[i]$  : vector with partially received event msg. from node  $i$  in element  $i$ 
3 //  $m_s$  : partially sent event msg.
4 //  $t_{slot}$  : start instant of this node's next CAN subslot
5 //  $d_{mw}$  : worst-case execution time of CAN emulation middleware
6 //  $g[i]$  : membership vector

7 // Transmission of event msgs.: update of outgoing state msg.  $s$  prior to node's own slot
8 when (  $t_{now} = t_{slot} - d_{mw}$  )
9    $s$  = empty msg.
10  while (space available in  $s$ )
11    if (  $m_s = \text{empty}$  )  $m_s \leftarrow$  protocol emulation // retrieve next event msg. to be sent
12    extract packet  $p$  from  $m_s$  // extract packet
13     $s \mid p$  // concatenate state msg.  $s$  with packet
14     $s \rightarrow$  TT network // no more space available in  $s$ 

15 // Reception of event msgs.: process incoming state msg.  $s$  from TT network
16 when ( (node  $i$  via TT network  $\rightarrow s_i = \{p_1, p_2, \dots\} \wedge (g[i] = 1)$  ) )
17   for every  $p_j \in s_i$  do
18      $m_r[i] \mid p_j$  // concatenate partially received msg. with packet
19     if  $m_r[i]$  completely assembled
20        $m_r[i] \rightarrow$  protocol emulation // forward assembled event msg. to next layer
21        $m_r[i] = \text{empty msg.}$  // start assembly of next msg.

```

Figure 5. Mapping between State and Event Messages through the Event Service

message from the network, the event service extracts the packets contained in the state message. The event service concatenates each packet with the partially assembled event message from the respective node. After the assembly of an event message is completed, the message is forwarded to the protocol emulation.

For error handling, the event service exploits the membership vector  $\vec{g}$  ( $g_i = 1$  if node  $i$  is correct, 0 otherwise) provided by the TT communication protocol. The state message corresponding to a particular sender node in the CNI is only processed, if the sender node is classified as correct according to the membership vector.

Note, that the realization of an EC as described above, i.e., by layering event messages on top of a service for the TT exchange of state messages supports different underlying communication protocols. As described in Section 3, most TT communication protocols provide at the controller-host interface state messages that are read or written by the communication system at a priori defined instants (i.e., temporal firewall interface). For example, in FlexRay the state messages exchanged in the so-called static segment [Fle05] are accessed by the application software via this temporal firewall interface. In TTP, all messages exchanged in a communication round are provided to the application software via the temporal firewall interface.

## 5. Protocol Emulation

Different types of message orderings (e.g., total order, FIFO order, causal order [HT94]) have been defined for distributed systems, because the interpretation of a message can depend on precedent messages. The protocol emulation focuses on the temporal order of the message receive instants, where the temporal order denotes the order of the instants on the timeline [BMN00].

The CAN protocol emulation is a middleware service that aims at the reuse of legacy applications with a minimum of redevelopment and retesting efforts. The CAN protocol emulation ensures that a VCN exhibits the same temporal order of the receive instants as a physical CAN network provided that the virtual network gets as input the same set of messages, in particular with identical transmission request instants. For this purpose, a *protocol emulation algorithm* is executed in every node to perform at run-time a simulation of a physical CAN network. This algorithm uses as inputs both messages for which a transmission has been requested by the local application (i.e., at the same node) and messages received via the event service from other nodes. The protocol emulation algorithm exploits the global timebase of the underlying TT protocol in order to capture in a timestamp the request instant of each message. The run-time simulation of a physical CAN network takes into account these timestamps of the request instants, as well as the message priorities, and the message lengths. Based on these inputs, the protocol emulation algorithm computes for each message the send instant when the message would have been sent on a physical CAN network. Messages are passed to the application in the order of ascending message send instants. Due to the non-preemptive nature of CAN, this strategy also ensures ascending message receive instants and thus the correct temporal message order. The protocol emulation algorithm is fully decentralized and runs in all nodes participating in the CAN emulation. It forms the middleware service for protocol emulation, which exchanges CAN messages with its adjacent service layers, namely the front-end and the event service.

The protocol emulation depends on the underlying event service in order to establish the temporal message order of a physical CAN network. For a precise emulation, the protocol emulation requires the event service to provide to each node the same bandwidth that would be available to the node in a physical CAN network (e.g., 500 kbps when emulating a high-speed automotive CAN bus). If no knowledge concerning the distribution of the bandwidth consumption is available, then the static allocation of communication resources via CAN subslots results in the need to provide at the event service a total bandwidth that is  $n$  times (where  $n$  is the number of nodes) as large as the bandwidth of the emulated physical CAN network. However, in many cases a priori knowledge about the maximum bandwidth consumption of an automotive node is available, e.g., expressed as a fraction of the overall bandwidth, which allows to considerably reduce the resource requirements of the VCN.

### Temporal Message Order of Event Channels

If the application software requests the transmission of a message  $m$  at instant  $t_{\text{request}}$ , the communication system will result in a delay until the complete message (i.e., the last bit of the CAN frame) has arrived at the receivers at instant  $t_{\text{receive}}$ . This communication system induced delay consists of the access delay  $d_{\text{access}}$  and the transmission delay  $d_{\text{transmission}}$ .  $d_{\text{transmission}}$  is determined by the bit rate of the network and the size of the message. The access delay  $d_{\text{access}}$  results from the media access protocol and the set of competing messages. On a physical CAN network, the access delay of a message comprises the remaining transmission delay of the currently transmitted message (since CAN is non-preemptive) and the transmission delays of all higher priority messages competing for bus access. The arbitration mechanism of the CSMA/CA media access

protocol exploits the dominant and recessive states of the bus for resolving contention. After an idle bus in the recessive state for at least 7 bit times, which is the duration of the end-of-frame delimiter, one or more nodes can start the transmission of a message. All CAN messages start with the transmission of the 11 or 29 bit identifier, the bits of which are subject to arbitration. During the transmission of each bit of the identifier, nodes compare the state of the bus with the sent bit. If a sent recessive bit is overwritten by a dominant bit, the respective node stops the message transmission, leaving the bus to the higher priority message. In this case, the access delay of the lower-priority message is increased by the transmission delay of the higher-priority one.

Without protocol emulation, ECs can exhibit a different temporal order of the receive instants compared to a physical CAN network due to the access delays caused by the underlying TT communication system. Firstly, messages from different nodes do not dynamically compete for the shared medium. All conflicts are resolved with a static schedule. Consequently, the message priorities (specified via the message identifiers) of two messages sent at separate nodes have no influence on the message send instants. The nodes do not share bandwidth and thus there is no dynamic decision on which message to transmit first. The send instant of each message is determined locally by the other transmission requests at this node. While this non-interference between nodes is beneficial from a fault isolation and composability perspective (i.e., a constructive integration of a distributed system), the missing contention between nodes also implies that legacy applications can perceive a different message order compared to a physical CAN network.

Secondly, although an EC accepts transmission requests at arbitrary instants, the dissemination of messages occurs only in the statically assigned CAN subslot of the respective node. After the transmission request of a message at a particular node, the message remains in a message queue until the respective node's CAN subslot occurs in the TDMA scheme. Since the CAN communication activities are not synchronized to this TDMA scheme, the access delays can include the duration of a complete TDMA round until the slot of the respective node reoccurs. The mapping from the continuous time of the message request instants to the discrete time of the start instants of CAN subslots corresponds to a sampling of CAN messages once every TDMA round. The access delay of a message depends on the instant of the transmission request relative to the start instant of the node's slot in the TT communication schedule.

### Phases of a CAN Message Transmission on a Virtual CAN Network

The transmission of a CAN message is requested at the front-end, which provides the API to the application software. Based on the information provided by the application (i.e., identifier, remote transmission request flag, 0-8 data bytes), the front-end builds a timestamped CAN message (step 1 in Figure 6). A timestamp  $t_{\text{request}}$  stores the transmission request instant w.r.t. the global time base established by the underlying TT communication protocol, i.e., the instant at which the CAN emulation at the sending node was handed over the message for being sent.

In a second step, the timestamped CAN message is broadcast to all other nodes of the distributed system via ECs. The timestamped CAN message informs the protocol emulation at other nodes about the message transmission request. In addition, the front-end forwards the timestamped CAN message to the local protocol emulation, i.e., the protocol emulation located at the node at which the transmission request has been issued. In conjunction with the consistency properties of the underlying TT communication service (which is the foundation for the ECs and the exchange of CAN messages), the CAN emulation at each node acquires a consistent set of messages that is used as input for message reordering through the simulation of the CSMA/CA protocol. The timestamped CAN messages arriving at a node's protocol emulation service are not immediately

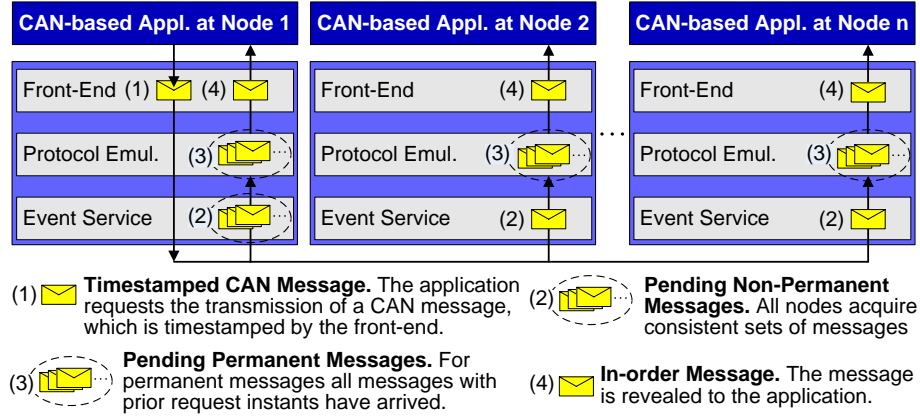


Figure 6. Phases of a CAN Message Transmission

revealed to the application, but remain pending in intermediate data structures until the correct message order can be established.

The timestamped CAN messages that arrive via ECs at the protocol emulation are initially *pending non-permanent messages* (step 2 in Figure 6). A message  $m$  with a transmission request instant  $t_{\text{request}}$  is non-permanent, if future messages, i.e., messages that have not yet arrived at the protocol emulation, can exhibit an earlier request instant than  $m$ . The notion of message permanence is based on the definition in [Kop97]. Since messages with an earlier request instant can precede  $m$  in the temporal message order, message  $m$  must become permanent before it can be used in the simulation.

After passing the *permanence test* described in Section 6, a pending message becomes permanent (step 3 in Figure 6). For a *permanent message*, it is ensured that all future messages will possess later request instants. The simulation determines the temporal message order based on the pending permanent message as its input. After a pending message has been sent in the simulation of the physical CAN bus (see Section 6), the message is denoted as *in-order*. As part of step 4 in Figure 6, the message is forwarded to the front-end to reveal it to the application software.

## Permanence Test

A message  $m_1$  is *permanent* at instant  $t_p$ , if it is known that no message  $m_2$  with an earlier or equal request instant ( $m_2.t_{\text{request}} \leq m_1.t_{\text{request}}$ ) can be received at a later instant  $t$  ( $t > t_p$ ) via an EC.

For determining permanence, one can exploit the fact that transmission request timestamps of messages received via an EC are monotonically increasing. The monotony of the transmission request timestamps results from the fact, that for every EC there is only a single sender (i.e., the protocol emulation layer at the respective node), which exclusively sends messages via the EC.

In order to determine the permanence of messages, the protocol emulation maintains a vector  $\vec{t}_{\text{latest}}$ , which contains a timestamp for every sender. The element  $i$  of this vector contains the message request instant of the most recent message received from sender  $i$ . Since the request instants of messages from a particular sender are monotonically increasing, the element associated with the sender in  $\vec{t}_{\text{latest}}$  represents a temporal bound for subsequent messages, i.e., all future timestamped CAN messages must contain a later request instant. Furthermore, the permanence

test exploits the global consistent membership vector  $\vec{g}$  ( $g_i = 1$  if node  $i$  is correct, 0 otherwise) provided by the TT communication protocol in order to exclude faulty nodes. In case a node is not in the membership (i.e.,  $g_i = 0$ ), the node's element in  $\vec{t_{latest}}$  is not used in the permanence test. Otherwise, a node with a crash failure would prevent the messages of other nodes from becoming permanent.

A sufficient condition for the permanence of a message is that its request instant is earlier than all temporal bounds for message request instants of correct nodes (from a total of  $n$  nodes) in the vector  $\vec{t_{latest}}$ :

$$\textbf{Permanence Test} \quad \bigvee_{i=1}^n (m.t_{\text{request}} < t_{\text{latest},i} \vee g_i = 0) \rightarrow m \text{ permanent}$$

## Message Ordering

The temporal ordering of messages occurs through a simulation of a physical CAN network at run-time, where simulated message transmissions represent the simulation steps. The current simulation time is specified by the instant  $t_{\text{idlestart}}$ .  $t_{\text{idlestart}}$  is a special instant that separates the messages which have been sent on the simulated CAN bus from those that have not.  $t_{\text{idlestart}}$  marks the beginning of idleness on the simulated CAN bus. The message transmissions before  $t_{\text{idlestart}}$  are already fixed, i.e., no later transmission requests can result in a modification of the sequence of message transmissions. Consequently,  $t_{\text{idlestart}}$  also separates the ordered messages from the non-ordered ones.

In case the simulation time lies before the minimum request instant of a future timestamped CAN message ( $t_{\text{idlestart}} < \min_i(t_{\text{latest},i})$  for all nodes  $i$ ) and one or more pending permanent messages are available, a simulation step can be taken. Out of the set of pending permanent messages, the protocol emulation chooses a message for the next simulation step based on the request instants and the message priorities. After the simulation step, the selected message becomes in-order and is transferred from the protocol emulation to the CAN front-end. Simulation steps are executed until no more pending permanent messages are available or a future timestamped CAN message can exhibit an earlier request instant than the current simulation time.

## Algorithm

The protocol emulation service executes the permanence test and the reordering of messages via the algorithm in Figure 7. This algorithm operates on two data structures, namely a heap of non-permanent pending messages and a heap of permanent pending messages. The elements of this heap are timestamped CAN messages. The primary key used for ordering in this heap is the sum of the message's request instant and the message's access delay. The secondary key is the message priority (identifier). The reason for selecting heaps as the data structures for pending messages is the need for the repeated retrieval of messages with the smallest request instant plus access delay and highest priority.

A message is inserted into the heap of non-permanent pending messages either when a message arrives from the network (i.e., the event service passes the messages to the protocol emulation in line 12) or after the application software has issued a transmission request at the front-end (line 8). In both cases, the front-end at the sending node has already set the transmission request timestamp of the timestamped CAN message.

The permanence test in line 15 is triggered by a change of the request instant vector  $\vec{t_{latest}}$  or the membership vector  $\vec{g}$ . The protocol emulation reads the message from the top of the heap of



---

```

1  // Variable description
2  //  $t_{\text{latest}}[]$  : vector containing most recent request instants
3  //  $g[]$  : membership vector
4  //  $H_{\text{nonperm}}$  : heap of non-permanent pending messages
5  //  $H_{\text{perm}}$  : heap of permanent pending messages
6  //  $t_{\text{idlestart}}$  : start of idle interval on simulated CAN bus
7  //  $t_{\text{request}}$  : transmission request instant of a message

8  when ( front-end  $\rightarrow m = \langle t_{\text{request}}, \text{id}, \text{data}, \text{this node}, 0 \rangle$  )
9       $t_{\text{latest}}[\text{this node}] = m.t_{\text{request}}$ 
10     insert  $m$  into  $H_{\text{nonperm}}$ 
11      $m \rightarrow \text{event service}$ 

12  when ( event service  $\rightarrow m = \langle t_{\text{request}}, \text{id}, \text{data}, \text{sender node}, 0 \rangle$  )
13       $t_{\text{latest}}[\text{sender node}] = m.t_{\text{request}}$ 
14      insert  $m$  into  $H_{\text{nonperm}}$ 

15  // Permanence test
16  when (  $H_{\text{nonperm}} \neq \emptyset \wedge \exists n \text{ for which } t_{\text{latest}}[n] \text{ or } g[n] \text{ has changed}$  )
17      // get non EC-permanent message with smallest timestamp
18      get message  $m = \langle t_{\text{request}}, \text{priority}, \text{data}, 0 \rangle$  from top of  $H_{\text{nonperm}}$ 
19      if (  $\forall n (t_{\text{latest}}[n] \geq m.t_{\text{request}}) \vee (g[n] = 0)$  )
20          //  $m$  is permanent
21          remove  $m$  from  $H_{\text{nonperm}}$ 
22           $m.t_{\text{request}} := \max(m.t_{\text{request}}, t_{\text{idlestart}})$ 
23          insert  $m$  into  $H_{\text{perm}}$ 

24  // Message Ordering
25  when (  $H_{\text{perm}} \neq \emptyset \wedge \nexists n t_{\text{latest}}[n] < t_{\text{idlestart}}$  )
26      get message  $m = \langle t_{\text{request}}, \text{priority}, \text{data}, d_{\text{access}} \rangle$  from top of  $H_{\text{perm}}$ 
27       $d_m$  = transmission duration of  $m$ 
28      remove  $m$  from  $H_{\text{perm}}$ 
29       $m \rightarrow \text{front-end}$ 
30       $t_{\text{idlestart}} = \max(m.t_{\text{request}}, t_{\text{idlestart}}) + d_m$ 
31      for every message  $n = \langle t_{\text{request}}, \text{priority}, \text{data}, d_{\text{access}} \rangle \in H_{\text{perm}}$  do
32           $n.d_{\text{access}} = \max(t_{\text{idlestart}} - n.t_{\text{request}}, 0)$ 
33      reorder  $H_{\text{perm}}$ 

```

---

Figure 7. Overview of Protocol Emulation Algorithm

non-permanent messages. If the permanence test gives a positive result, the message is removed from the heap of non-permanent and inserted into the heap of permanent pending messages. The retrieval of a message from the top of the heap with the subsequent evaluation of the permanence condition proceeds until the heap becomes empty or the permanence test fails. As soon as the permanence test fails for a message, the permanence checking is finished, because the ordering of the heap ensures that the permanence test also fails for all other messages in the heap.

A simulation step in the simulation of the physical CAN network is triggered by the availability of permanent pending messages (see line 23) and a simulation time that is earlier than all elements of the request instant vector  $\overrightarrow{t_{\text{latest}}}$ . The protocol emulation removes the message  $m$  from the

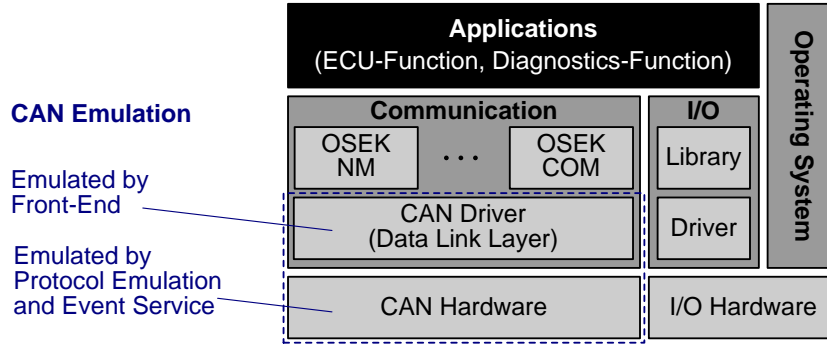


Figure 8. Model of an Automotive ECU

top of the heap of permanent pending messages and forwards the  $m$  to the front-end (invocation of a message push function of the front-end). Subsequently, the simulation time and the access delays of the pending messages are updated using the transmission duration  $d_{\text{transmission}}$  of message  $m$ , which depends on the length of  $m$ , the identifier type, the bandwidth of the emulated CAN network, and the bit stuffing overhead.

Since the primary keys of messages in  $H_{\text{perm}}$  have changed (i.e., different access delays), it is necessary to perform a reordering of the heap. In general, multiple messages will now be ordered by the secondary key (message identifier). For these messages, the selection for transmission in the protocol emulation will occur based on the message priority, which corresponds to the media access control strategy of CAN.

## 6. Front-End

In present-day automotive ECUs, CAN drivers establish generic APIs towards the underlying CAN communication system. These APIs abstract from any particular CAN controller chip and provide to the application software a generic interface that abstracts from low-level details (e.g., register set of a particular CAN controller, message buffer configurations like Full-CAN and Basic-CAN). CAN-based APIs facilitate the separation of the hardware of an ECU from its embedded software, which has been identified as a key requirement for the reuse of automotive software [HKK04].

The front-end of the CAN emulation middleware realizes the API of the HIS/VectorCAN driver from the OEM Initiative Software [Vol03]. The reason for choosing HIS/VectorCAN driver over other higher protocols and APIs (e.g., CANopen [CAN05], DeviceNet [NSM94]) is the wide use and support in the automotive domain by vehicle manufacturers Audi, BMW, DaimlerChrysler, Porsche, and Volkswagen<sup>1</sup>. Nevertheless, due to the modular structure of the CAN emulation with its three layers, only a single layer – the Front-End – needs to be replaced in order to establish a different CAN API. Figure 8 gives an overview of the standard software core of an automotive ECU [HKK04]. While the event service and the protocol emulation take over the responsibility to emulate the CAN hardware, the establishment of the CAN driver is within the responsibility of the front-end.

<sup>1</sup>[www.automotive-his.de](http://www.automotive-his.de)

In compliance with the specification of the HIS/VectorCAN driver, the front-end of the CAN emulation middleware offers to the application one or more *message handles*, each being assigned to a specific CAN message, which is similar to a full CAN message buffer, or to a range of messages which is similar to a basic CAN message buffer. A message handle is either a transmit object or a receive object and incorporates a CAN identifier, a data length code, a pointer to the CAN frame data, and pointers to callbacks. Based on filter masks, each message handle is associated with a subset of the identifiers representing the CAN namespace. Thereby, the application can selectively decide which messages are accepted, i.e., stored in message buffers and triggering callbacks.

Callbacks are sporadic computational activities, which are registered at the front-end. The occurrence of a triggering event causes the activation of the callback through the front-end. The front-end supports the following types of callbacks:

- **Error handling callbacks.** In these callbacks, the application can react to the bus off state, to fatal errors, overrun errors, and syntactically incorrect CAN messages (e.g., invalid data length code).
- **Transmission callbacks.** The transmission callbacks include a pretransmit function, which can be used for setting the data bytes of an outgoing CAN message, as well as a confirmation function that serves as an acknowledgment of a successful message transmission.
- **Reception callbacks.** These callbacks comprise reception notification functions, e.g., for user-defined identifier ranges and specific message handles.

## Transmission Path

The CAN-based application initiates the transmission process by invoking a *transmit operation* at the API provided by the front-end. The application identifies the CAN message that shall be sent via a message handle. The invocation of the transmit operation represents a transmission request from the application, which triggers the activities comprising the transmission path depicted in Figure 9.

After being invoked by the transmit operation, the front-end first checks whether the transmission path is enabled. In the next step, the front-end searches for the entry in the transmit data structure that matches the handle specified by the application. If a *pre-transmit callback* is registered for the handle, it is now called in order to give the CAN-based application the ability to execute application-specific code prior to the message transmission. In particular, the CAN-based application is passed a reference to the message handle, thus allowing the application to modify the contents of the CAN message (identifier, data length code, data bytes). The front-end, then, passes the message to the protocol emulation in order to broadcast the message via the VCN and invokes the *transmission-start callback*.

The front-end exploits the membership information provided by the underlying TT communication protocol in order to determine whether a sent message has been successfully received by all correct nodes. The membership information serves as a substitute for the acknowledgment bit of a physical CAN network. A successful transmission from a node is indicated through a set membership bit of the node a TDMA round after a message has been sent. In this case, the front-end invokes the *confirmation callback*.

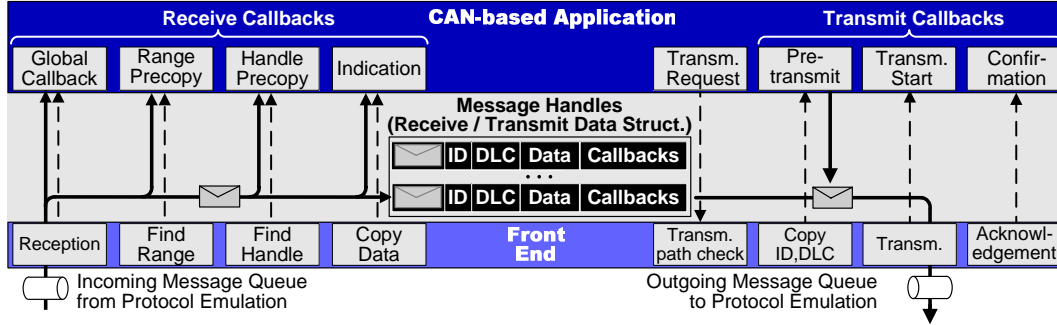


Figure 9. Message Transmission and Reception Path

## Reception Path

In contrast to the transmission process, the reception process is triggered by a message received from a VCN. The front-end checks upon each invocation whether incoming messages are present in the queue forming the interface towards the protocol emulation (see Figure 9). If a message is present, the front-end invokes the *global callback*, which is a common callback function for all messages. The front-end, then, determines if the message identifier matches one of four identifier ranges that are defined by statically set bit masks. Each of the four ranges possesses a *range-precopy callback* function, which is called in case of an identifier match.

In the next step, the front-end determines whether the receive data structure contains a message handle matching the identifier of the received message. Thereby, the front-end implements message filtering, because the message is discarded when there is no match with a handle in the receive data structure. After the execution of the corresponding *handle-precopy callback*, the front-end writes the contents of the message (identifier, data length code and data bytes) into the message handle's entry in the receive data structure. Finally, an *indication callback* in the CAN-based application is called in order to allow the CAN-based application to process the new message in the receive data structure.

## 7. Results and Discussion

A prototype implementation of a VCN on top of a physical network running the Time-Triggered Protocol (TTP) has served as the validation platform [TTT02a]. The underlying TTP network has provided two redundant channels with a bitrate of 25 Mbps. The three layers of the CAN emulation have been subject to test applications and real-world automotive communication matrices provided by a vehicle manufacturer. The event service has proven effective to handle aperiodic and sporadic message transmissions with comparable latencies as a physical CAN network and superior bandwidth. With enabled protocol emulation, the VCN has also exhibited the exact same temporal message order as a physical CAN network. Finally, the conformance of the API with the HIS/VectorCAN driver specification has been validated with the diagnostic protocol stack from a vehicle manufacturer.

In order to compare the behavior of the VCN with a physical CAN network, a MATLAB/Simulink simulation of a physical CAN network has served as a reference point. The validation of the event service and protocol emulation has occurred through test applications executed in both the implementation of the VCN and a MATLAB/Simulink framework. The test applications

have performed transmission requests at predefined instants and included sequence numbers and timestamps denoting the transmission request instants in the broadcasted CAN messages.

Furthermore, this sections discusses the effects of different TT communication schedules (i.e., different duration of communication rounds) on the delay of messages exchanges via a VCN.

## Measurement Framework – Virtual CAN Network

The measurement framework employs an implementation of a VCN on a cluster with four TTP MPC855 single board computers [TTT02b] as the nodes. Each node is equipped with an MPC855 PowerPC from Freescale clocked with 80 MHz and contains the C2 (AS8202) TTP communication controller. The TTP MPC855 single board computer uses the embedded real-time Linux variant RTAI [BBD<sup>+</sup>00] as the operating system, combining a real-time hardware abstraction layer with a real-time application interface for making Linux suitable for hard real-time applications [RTA00]. The CAN emulation middleware has been realized via a time-triggered RTAI-task, which has been periodically invoked in each communication round by a control signal from the communication controller. For this task a worst-case execution time of 80  $\mu$ s has been observed in the measurements.

A distributed measurement application uses off-line computed message transmission request tables as inputs for the VCN. The measurement application at every sender node accesses the VCN and requests message transmissions at the instants specified in the request table (see Figure 10). The table also determines the length and identifier of each transmitted message. The data area of the CAN message contains a message index, which uniquely identifies a particular message transmission request along with the node from which the message originated.

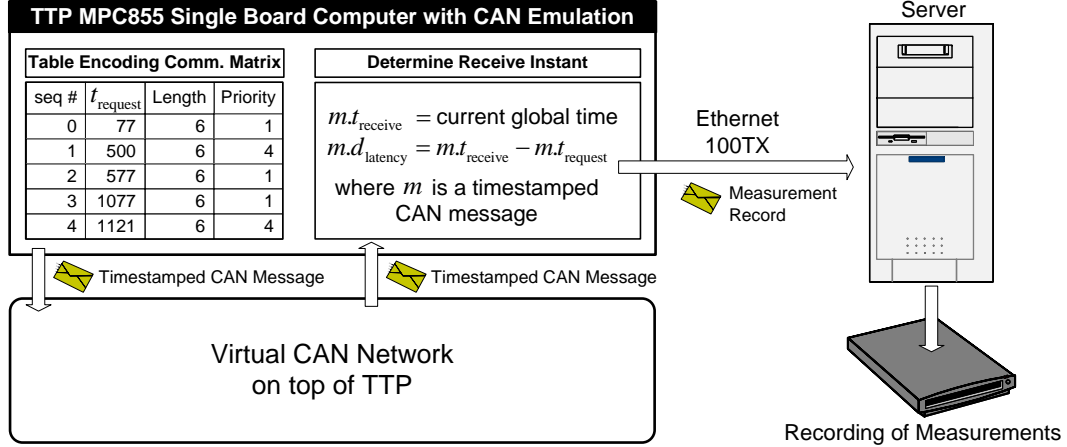


Figure 10. Measurement Framework

Whenever a message  $m$  is received, the respective node assigns a timestamp  $m.t_{\text{receive}}$  to the received message and computes the transmission latency ( $m.d_{\text{latency}} = m.t_{\text{receive}} - m.t_{\text{request}}$ ) by exploiting the global timebase provided by TTP.  $m.t_{\text{request}}$  denotes the point in time at which the message needs to be sent according to the off-line computed message transmission request tables. The transmission latency  $m.d_{\text{latency}}$  incorporates queuing delays of the VCN at the sender, latencies of the underlying network and execution times of the CAN middleware. A measurement record is constructed with the index of the sending node, the sequence number of the message, the

receive instant  $m.t_{\text{receive}}$ , and the transmission latency  $m.d_{\text{latency}}$ .

$$\langle \text{sender node, msg. sequence number, } m.t_{\text{receive}}, m.d_{\text{latency}} \rangle \quad (1)$$

This measurement record is stored in an Ethernet message and transferred to a workstation. The workstation collects the measurement records from the TTP MPC855 single board computers for a later analysis.

### Simulation Framework – Physical CAN Network

The simulation framework based on the MATLAB/Simulink toolbox *TRUETIME* provides information about the behavior of a physical CAN network, when provided with a particular message pattern as input. *TRUETIME* [HCA02] is a MATLAB/Simulink-based simulation toolbox for real-time control systems. *TRUETIME* supports the simulation of the temporal behavior of tasks in a host computer, as well the simulation of the timing behavior of communication networks. For this purpose, it offers two Simulink blocks: a computer block and a network block. In the framework, a *TRUETIME* network block has been parameterized with the CSMA/CA medium access control protocol of CAN, while the nodes are modeled by *TRUETIME* computer blocks. In every node, a task is executed that transmits messages according to the message transmission request table as described in the measurement framework. At the points in time specified in this table, the task passes CAN messages to the *TRUETIME* network. Each CAN message is assigned the priority and length as specified in the table. One of the nodes also hosts a reception task and writes the message sequence numbers and the transmission latencies into a file for a later analysis.

### Results – Comparison of Physical and Virtual CAN Networks

For the validation of the CAN emulation, a real-world automotive communication matrix has been used as input for both the implementation of VCNs and a simulation of a physical CAN network. Based on the communication matrix, the message transmission request tables have been computed in order to parameterize the distributed measurement application of the VCN and the sender tasks of the *TRUETIME* computer blocks. The communication matrix originates from a powertrain network and consists of 102 periodic messages. The message periods range from 3.3 ms to 1 s, the number of data bytes is between 2 and 8 bytes. Messages comply with the standard-CAN format [Bos91] and contain 47 control bits, thereby resulting in a total message size between 47 and 111 bits. The overall network bandwidth required for the exchange of these messages is 300 kbps.

At the underlying TT communication schedule, the communication schedule provides CAN slots for four nodes with a communication round length of 320  $\mu\text{s}$ . The CAN slot of each node has a size of 64 bytes. In the simulation, the bandwidth of the network block has been set to 500 kbps.

The simulation results for the automotive message sets are depicted on the left-hand side of Figure 10. The x-axis represents the message transmission latencies. The message identifiers are distinguished along the y-axis. These identifiers range from 80 to 2004 and denote the message priority. Larger CAN identifiers correspond to lower message priorities. The distance along the z-axis represents the number of messages with a given message priority and transmission latency. Figure 10 shows that high priority messages make up for a large amount of the overall bandwidth. The third of messages with the highest priority makes up for 49% of exchanged messages. The simulation results also demonstrate the high average performance of CAN. 97% of all message transmissions possess transmission latencies below 1 ms. However, transmission latencies vary

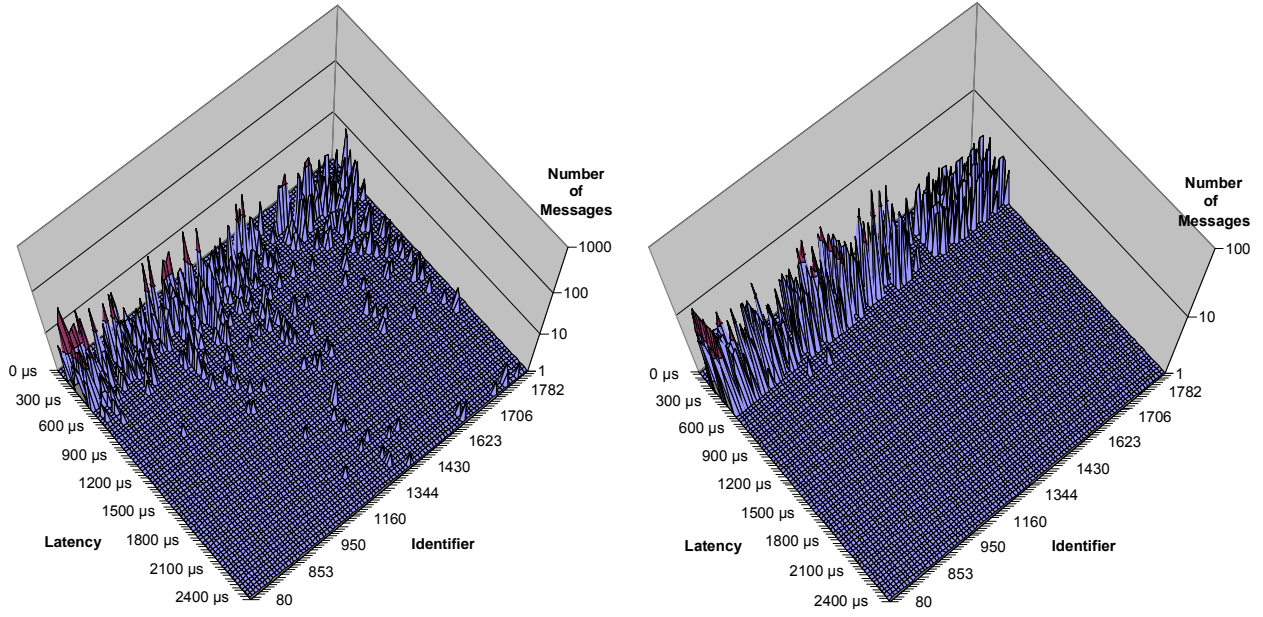


Figure 11. Simulation Results (left) and Measurement Results (right) with Logarithmic Scaling of z-axis

considerably. The logarithmic scaling of the z-axis in Figure 10 emphasizes the rare cases, in which transmission latencies significantly differ from the average values.

The right-hand side of Figure 10 depicts the measured transmission latencies for the automotive message sets in the implementation of the event service (i.e., before protocol emulation). The observed worst-case latency is  $608\ \mu\text{s}$ , i.e., significantly lower compared to the  $3165\ \mu\text{s}$  of the physical CAN network. The best-case latency is  $147\ \mu\text{s}$ . The average message transmission latency of the measurements is  $323.51\ \mu\text{s}$ .

The correct behavior of the protocol emulation has been demonstrated via the sequence numbers contained in the measurement records. Both the measurement framework and the simulation frameworks assign sequence numbers to received messages in order to capture the temporal message order. In test runs without protocol emulation, the message sequences have been different for the physical CAN network and the VCNs. With enabled protocol emulation, the sequence numbers in the measurement records of the VCN have shown the same message order as in the simulation of the physical CAN network.

However, there is a fundamental trade-off between improving the worst-case latencies in comparison to a physical CAN network and the establishment of the correct temporal message order. With enabled protocol emulation, the worst-case latency of the VCN is also  $3.1\ \text{ms}$ , since the CSMA/CA protocol is emulated. Although a CAN message is received earlier via the event service, the forwarding to the front-end is delayed until the message wins in the simulated arbitration process. For this purpose, a designer needs to decide whether he prefers a superior temporal performance or enables protocol emulation in order to ensure the correct behavior of legacy software without redevelopment efforts.



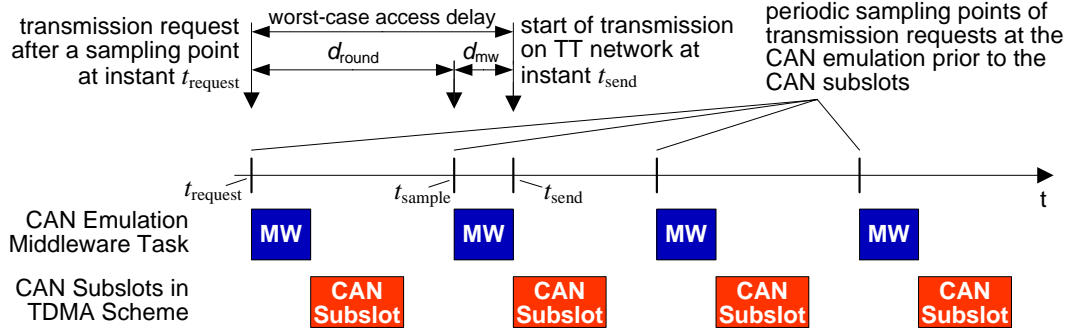


Figure 12. Worst-Case Access Delay Induced by Mapping to the Time-Triggered Network

## Results – Integration of CAN-based Diagnostic Software Stack

The implementation of the CAN emulation has been used to integrate the diagnostic software stack of an automotive company into nodes of a TT system, including a transport layer and a diagnostic event handler as part of KWP2000 [Int99]. The diagnostic software stack builds on top of the API of the HIS/VectorCAN driver, thus exploiting the services of the front-end in the CAN emulation.

KWP2000 is used in maintenance mode in order to retrieve diagnostic information, such as breakdown log entries. Breakdown log entries are generated by the OBD system in a node computer, once an error is detected (e.g., by the Built-In Self Test (BIST) or by application-specific plausibility checks). At the service station the mechanic uses a diagnostic testing device (e.g., VAS tester) to retrieve breakdown log entries and determine the Diagnostics Trouble Codes (DTC).

In conjunction with a CAN/TTP gateway implemented on a TTP node [TTT04], it is possible to diagnose the TTP network with a standard diagnostic computer (VAS Tester) as presently used in repair stations. For example, using the VAS tester the diagnostic information provided by C2 controller can be displayed (e.g., membership vector of the TTP nodes).

## Delay of Event-Triggered Message Exchanges Induced by Mapping to the Time-Triggered Network

In an integrated system for TT and CAN-based applications, the duration of a communication round on the TT network is an application-specific parameter that contributes significantly to the end-to-end delay of messages exchanged via a VCN. As depicted in Figure 12, the application requests the transmission of a message at the request instant  $t_{\text{request}}$ . This action causes an update of the interface data structures provided by the CAN emulation middleware, which contain now a new message that needs to be transmitted via the VCN. After the request instant the access delay (see also Section 5) follows before the transmission of the CAN message is started on the TT network at instant  $t_{\text{send}}$ . Since CAN-based applications are not synchronized to the TDMA scheme of the TT network, the access delay depends on the request instant relative to the start instant of the next CAN subslot. The CAN emulation middleware periodically samples the transmission requests prior to each CAN subslot, thus incurring a worst-case delay of a complete communication round in case a message transmission is requested immediately after a sampling point of the CAN emulation middleware. In this case, which is depicted in Figure 12, the sampling of the message transmission request (at instant  $t_{\text{sample}}$ ) is delayed until the CAN emulation is activated



again for the next CAN subslot. Therefore, the worst-case access delay comprises the sum of the communication round duration  $d_{\text{round}}$  and the worst-case execution time  $d_{\text{mw}}$  of the CAN emulation middleware. The access delay jitter is equal to the duration of a communication round, while the average access delay is  $d_{\text{mw}} + d_{\text{round}}/2$ , when assuming a uniform distribution of the request instants within a communication round.

Based on this analysis, the effects of different TT communication schedules can be quantified. In many practical systems, a communication round length of  $320\ \mu\text{s}$  as used in the prototype implementation can be difficult to attain due to larger numbers of nodes and application software requiring longer execution time slots (e.g., several ms). For example, assuming a communication round length of 5 ms, both the worst-case access delay and the access delay jitter would be extended by 4.68 ms in comparison to the  $320\ \mu\text{s}$  schedule. The average access delay would be increased by 2.34 ms due to an average delay of 2.5 ms before the next sampling point (instead of  $160\ \mu\text{s}$  in case of the  $320\ \mu\text{s}$  schedule).

## 8. Conclusion

With the introduction of TT communication protocols in safety-related and safety-critical computer systems of upcoming car series, the reuse of CAN-based legacy application in TT computer systems becomes of high economic relevance. The CAN emulation provides a basis for designers who intend to reuse CAN-based applications despite the migration to a TT platform. In addition to the ability to retain investments in existing software, such a migration permits the replacement of physical CAN networks through event channels. As overlay networks on the TT communication service, the resulting *virtual CAN networks* enable significant cost and reliability benefits through the reduction of connectors and wiring. In order to minimize the redevelopment efforts for CAN-based legacy applications, it is important to provide to applications the same interfaces to the underlying communication system as in a physical CAN system. In particular, the correct behavior of many legacy applications depends on the temporal properties (bandwidth, latencies, message order) of the emulated CAN communication service. For this reason, virtual CAN networks as described in this paper not only support ET on-demand communication activities on top of a TT communication protocol, but also emulate the CSMA/CA media access control protocol of CAN in order to ensure the same temporal message order as a physical CAN network. By exploiting the higher native bitrate of the underlying TT network, virtual CAN networks can exceed physical CAN network with respect to bandwidth and latencies. The ability of virtual CAN networks to handle the communication needs of automotive applications has been demonstrated in a validation framework employing a real-world automotive communication matrix and a diagnostic protocol stack.

## Acknowledgments

This work has been supported in part by the European IST project DECOS (IST-511764) and the European IST project ARTIST2 (IST-004527).

## References

- [AG03] A. Albert and W. Gerth. Evaluation and comparison of the real-time performance of CAN and TTCAN. In *Proc. of 9th International CAN Conference*, Munich, 2003.
- [BBD<sup>+</sup>00] D. Beal, E. Bianchi, L. Dozio, S. Hughes, P. Mantegazza, and S. Papacharalambous. RTAI: Real-Time Application Interface. *Linux Journal*, April 2000.

- [BMN00] P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Computing Surveys (CSUR)*, 32(1):12–42, 2000.
- [Bos91] Robert Bosch GmbH, Stuttgart, Germany. *CAN Specification, Version 2.0*, 1991.
- [CAN05] CANopen application layer and communication profile v4.0.2. Technical report, CiA DS 301, March 2005.
- [Cri91] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.
- [Fle05] FlexRay Consortium. BMW AG, DaimlerChrysler AG, General Motors Corporation, Freescale GmbH, Philips GmbH, Robert Bosch GmbH, and Volkswagen AG. *FlexRay Communications System Protocol Specification Version 2.1*, May 2005.
- [For04] Analysis of the european automotive in-vehicle network architecture markets. Technical report, Frost & Sullivan, October 2004.
- [Fre05] Freescale Semiconductor. MFR4200 datasheet FlexRay communication controllers. Technical report, August 2005.
- [HCA02] D. Henriksson, A. Cervin, and K.E. Arzen. Truetime: Simulation of control loops under shared computer resources. In *Proceedings of the 15th IFAC World Congress on Automatic Control*, Barcelona, Spain, July 2002. Department of Automatic Control, Lund Institute of Technology.
- [HD93] K. Hoyme and K. Driscoll. SAFEbus. *IEEE Aerospace and Electronic Systems Magazine*, 8:34–39, March 1993.
- [Hei03] H.D. Heitzer. Development of a fault-tolerant steer-by-wire steering system. *Auto Technology*, 4:56–60, April 2003.
- [HKK04] B. Hardung, T. Kølzow, and A. Krüger. Reuse of software in distributed embedded automotive systems. In *EMSOFT '04: Proceedings of the fourth ACM international conference on Embedded software*, pages 203–210, New York, NY, USA, 2004. ACM Press.
- [HT94] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca NY, USA, May 1994.
- [IEE02] IEEE Std. 1588 - 2002 IEEE Standard for a precision clock synchronization protocol for networked measurement and control systems. Technical report, IEEE Instrumentation and Measurement Society, November 2002. ISBN: 0-7381-3369-8.
- [Int99] International Organization for Standardization. *Keyword Protocol 2000, ISO 14230*, 1999.
- [KGR91] H. Kopetz, G. Granstedt, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. *Dependable Computing for Critical Applications (A. Avizienis and J. C. Laprie, Eds.), pp.411-29*, Springer-Verlag, Wien, Austria, 1991.
- [KN97] H. Kopetz and R. Nossal. Temporal firewalls in large distributed real-time systems. *Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '97)*, 1997.
- [Kop97] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.
- [LH02] G. Leen and D. Heffernan. Expanding automotive electronic systems. *Computer*, 35(1):88–93, January 2002.
- [LS04] J. Leohold and C. Schmidt. Communication requirements of future driver assistance systems in automobiles. In *Proceedings of the IEEE International Workshop on Factory Communication Systems*, pages 167–174, September 2004.
- [NSM94] D. Noonan, S. Siegel, and P. Maloney. DeviceNet application protocol. In *Proceedings of the 1st International CAN Conference*, Mainz, Germany, 1994.
- [OSE05] OSEK/VDX. Operating system – version 2.2.3. Technical report, February 2005.
- [PA00] P. Pedreiras and L. Almeida. Combining event-triggered and time-triggered traffic in FTT-CAN: Analysis of the asynchronous messaging system. In *Proceedings of 3rd IEEE International Workshop on Factory Communication Systems*, September 2000.
- [RTA00] RTAI Programming Guide, Version 1.0. Dipartimento di Ingegneria Aerospaziale Politecnico di Milano (DIAPM), Italy, September 2000. Available at <http://www.rtai.org>.
- [Rus01] J. Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, September 2001.

- [SKR97] P. Lincoln S. Katz and J. Rushby. Low-overhead time-triggered group membership. In *Proc. of 11th International Workshop on Distributed Algorithms, Lecture Notes in Computer Science*, volume 1320, pages 155–169. Springer-Verlag, 1997.
- [SM99] J. Swingler and J.W. McBride. The degradation of road tested automotive connectors. In *Proc. of the 45th IEEE Holm Conference on Electrical Contacts*, pages 146–152, Pittsburgh, PA, USA, October 1999. Dept. of Mech. Eng., Southampton Univ.
- [TBW<sup>+</sup>00] T. Fahrner, B. Müller, W. Dieterle, F. Hartwich, and R. Hugel. Time-triggered CAN - TTCAN: Time-triggered communication on CAN. In *Proc. of 6th International CAN Conference (ICC6)*, Torino, Italy, 2000.
- [TTT02a] TTTech Computertechnik AG, Schönbrunner Strasse 7, A-1040 Vienna, Austria. *Time-Triggered Protocol TTP/C – High Level Specification Document*, July 2002.
- [TTT02b] TTTech Computertechnik AG, Schönbrunner Strasse 7, A-1040 Vienna, Austria. *TTP Monitoring Node – A TTP Development Board for the Time-Triggered Architecture*, March 2002.
- [TTT02c] TTTech Computertechnik AG, Schönbrunner Strasse 7, A-1040 Vienna, Austria. *TTP/C Controller C2 Controller-Host Interface Description Document, Protocol Version 2.1*, November 2002.
- [TTT04] TTTech Computertechnik AG, Schönbrunner Strasse 7, A-1040 Vienna, Austria. *TTP Development Cluster Rŕ- The Complete TTP System*, 2004.
- [Vec05] Vector Informatik GmbH. CANalyzer and DENalyzer 5.2 – the tools for comprehensive network analysis. Technical report, Stuttgart, 2005.
- [Vol03] Volkswagen AG. HIS/VectorCAN driver specification 1.0. Technical report, Berliner Ring 2, 38440 Wolfsburg, August 2003.



# DETECTION OF OUT-OF-NORM BEHAVIORS IN EVENT-TRIGGERED VIRTUAL NETWORKS

*Proceedings of the 5th IEEE International Conference on Industrial Informatics, Volume 1, pages 541–546. Vienna, Austria, July 2007.  
(best paper award)*

Roman Obermaisser  
Vienna University of Technology  
Real-Time Systems Group  
romano@vmars.tuwien.ac.at

Philipp Peti  
Vienna University of Technology  
Real-Time Systems Group  
peti@vmars.tuwien.ac.at

**Abstract** The DECOS system architecture integrates time-triggered and event-triggered control for combining the benefits of both paradigms. For applications based on event-triggered control, this architecture establishes event-triggered virtual networks as overlay networks on top of an underlying time-triggered physical network. In the scope of this paper, we show that the underlying time-triggered network significantly improves the accuracy of the error detection mechanisms in comparison to event-triggered architectures used today (e.g., based on Controller Area Network in the automotive industry). We introduce a framework with detectors for out-of-norm behavior, which are distributed across the nodes of the distributed real-time system. The detectors produce diagnostic messages augmented with information about the location and time of the detection event. Due to the fault isolation and the global time base of the underlying time-triggered network, the additional spatial and temporal information (besides the value domain) is available in the diagnostic messages and forms a meaningful input to a subsequent analysis process. The proposed framework manages the inherently imprecise temporal specifications of event-triggered application subsystems by correlating diagnostic messages along value, space and time. Thereby, a discrimination between a correct behavior of the computer system (e.g., triggered by rare conditions in the environment) and different fault classes (e.g., design faults, physical faults) becomes feasible. Also, the paper describes an implementation of the framework based on the Time-Triggered Protocol and explains the specification of the detectors and the analysis process using timed automata.

## 1. Introduction

Advances in computer and communication technologies have made it feasible to extend the application of embedded computer systems to more and more safety-critical applications, e.g., in the automotive and avionic domain. Due to the many different and, partially, contradicting requirements, there exists no single model for building the communication system of a distributed computer system that interacts with a physical environment. Well-known tradeoffs are predictability

versus flexibility, and resource adequacy versus best-effort strategies [Kop97]. Thus, the chosen system model depends significantly on the requirements of the application.

It has been recognized that communication systems fall into two general categories with corresponding strengths and deficiencies: time-triggered and event-triggered control. Event-triggered protocols (e.g., CAN [Bos91]) offer flexibility and resource efficiency, while time-triggered protocols (e.g., TTP [KG94], FlexRay [Fle05]) excel with respect to predictability, composability, error detection and error containment. In [OPHS06] an integrated architecture (called DECOS architecture) has been introduced that brings together the benefits of both paradigms. This architecture realizes event-triggered virtual networks as overlay networks on top of an underlying time-triggered physical network. Through the support for both time-triggered and event-triggered communication activities, the integrated architecture is suitable for building mixed-criticality systems and the reuse of legacy applications. For time-triggered applications, the DECOS architecture exploits the knowledge concerning the predetermined points in time of the periodic message transmissions for error detection and the establishment of membership information [Ade03]. These error detection services are not only employed for realizing the architectural fault isolation and fault tolerance services, but also provided as feedback to applications in order to control application level fault-tolerance.

This paper goes beyond these error detection mechanisms that are restricted to time-triggered applications only. As part of the integrated DECOS architecture, we present a solution for improved error detection services of event-triggered application subsystems. The challenge for detecting errors in event-triggered application subsystems is the inherent impreciseness of temporal specifications in this control paradigm [Obe04]. In general, event-triggered control is chosen due to its flexibility. Since the points in time of communication activities need not be fixed at design time, the temporal behavior of communication activities can be determined on-demand by the application software. Thus, modifications of application software do not necessarily require a reconfiguration to the communication system and communication resources are only used when events occur. However, this benefit of flexibility has adverse implications for the ability of error detection [Kop97, p. 164]. Error detection is based either on redundant computations or an a priori knowledge, while flexibility involves the limiting of a priori knowledge.

To overcome this difficulty of limited a priori knowledge, we provide a solution to error detection for event-triggered application subsystems by gathering and correlating information about improbable behavior denoted as *out-of-norm behavior*. Out-of-norm behavior is detected through the execution of deterministic timed automata, which access the consistent distributed state of the DECOS architecture as defined with respect to a global sparse time base [Kop92].

The timed automata generate diagnostic messages as indications of out-of-norm behavior augmented with information concerning space, time, and value. The diagnostic messages are sent to an analysis process that concentrates information about multiple out-of-norm behavior occurrences in order to conclude whether an error has occurred. By augmenting each detection event with information about the locality and the global point in time of the detected out-of-norm behavior, the correlation process is significantly simplified. In particular, the meaningfulness of the spatial information is ensured through the fault isolation of the DECOS architecture [OP05]. The architecture's fault isolation services prevent the propagation of faults, thus allowing to pinpoint the exact location of a fault. Also, the clock synchronization service of the DECOS architecture is the basis for correlating the timestamps assigned to different out-of-norm behavior detections.

The solution for error detection, which is proposed in this paper, is a generic architectural service and can be parameterized to adapt it to specific applications. Since control on the sending and receiving instants is under the sphere of control of the event-triggered application and not the com-

munication system, the detectors for out-of-norm behavior need to be parameterized according to the communication model behind the application. This allows not only to detect deviations from the specification, but also to gather facts whether the underlying assumptions behind the communication model hold in reality. For instance, unanticipated customer behavior (e.g., playing with the window lifter button) may impose serious problems to the functionality of the system.

This paper is structured as follows. Section 2 gives an overview of the DECOS architecture. As part of this architecture, the detection of out-of-norm behavior will be discussed in Section 3. Section 4 outlines possible applications of the proposed solution. An implementation of the framework is the focus of Section 5. The paper finishes with a discussion in Section 6.

## 2. DECOS Architecture

The DECOS architecture [OPHS06] offers a framework for the development of distributed embedded real-time systems integrating multiple application subsystems with different levels of criticality and different requirements concerning the underlying platform.

Structuring rules guide the designer in the decomposition of the overall system both at a functional level and for the transformation to the physical level. The services of the real-time computer system are divided into a set of nearly-independent DASs. Each DAS is further decomposed into smaller units called *jobs*. A job is the basic unit of work and exploits a *virtual network* [OP05] in order to exchange messages with other jobs and work towards a common goal. A *virtual network* is the encapsulated communication system of a DAS. All communication activities of a virtual network are private to the DAS, i.e., transmissions and receptions of messages can only occur by jobs of the DAS unless a message is explicitly exported or imported by a gateway. Furthermore, a virtual network exhibits predefined temporal properties that are independent from other virtual networks.

A *port* is the access point between a job and the virtual network of the DAS the job belongs to. Depending on the data direction, one can distinguish input ports and output ports. In addition, we classify ports into state ports and event ports depending on the information semantics [Kop97] of send or received message.

In addition, the DECOS integrated architecture aims at offering to system designers generic architectural services, which provide a validated stable baseline for the development of applications (see Figure 1). The architectural services consist of *core services* and *high-level services*. The core services include a predictable time-triggered message transport, fault tolerant clock synchronization, and strong fault isolation. Any architecture that provides these core services can be used as a base architecture [Rus01] for the DECOS integrated distributed architecture. An example of a suitable base architecture is the Time-Triggered Architecture (TTA) [KB03]. On top of the core services, the DECOS integrated architecture realizes high-level architectural services, which are DAS-specific and constitute the interface for the jobs to the underlying platform. Among the high-level services are gateway services, virtual network services, encapsulation services, and error detection services. On top of the time-triggered physical network, different kinds of virtual networks are established and each type of virtual network can exhibit multiple instantiations (see Figure 1). The encapsulation services control the visibility of exchanged messages and ensure spatial and temporal partitioning for virtual networks in order to obtain error containment.

## 3. Detection of Out-of-Norm Behavior

This section describes the mechanisms for the detection of out-of-norm behavior, which are part of the high-level architectural service for diagnosis in the DECOS architecture. After refining the

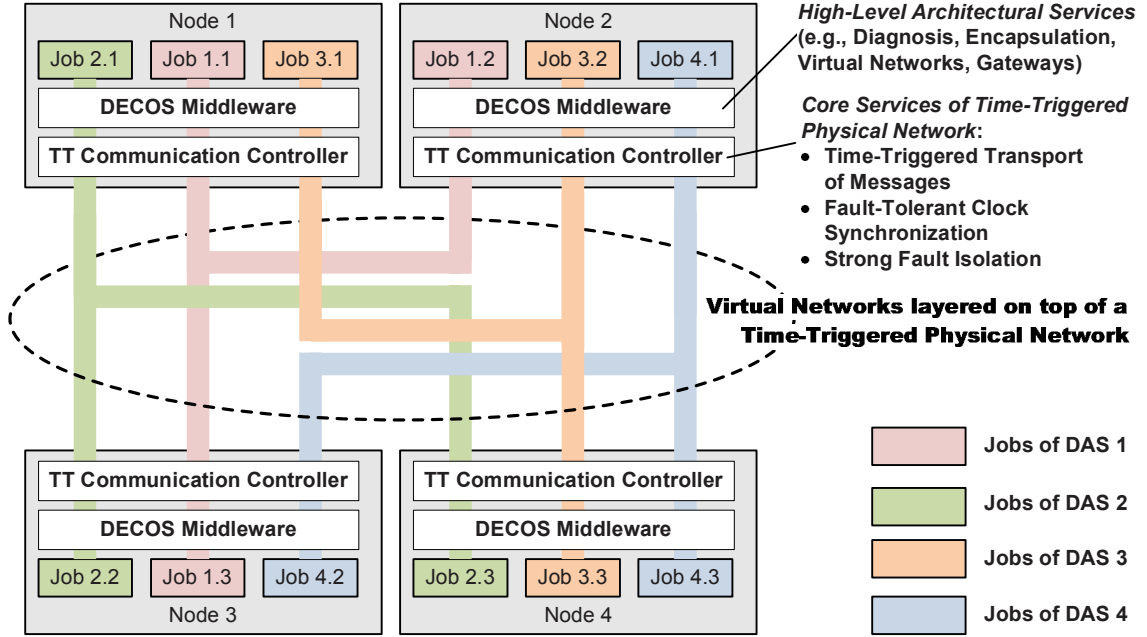


Figure 1. DECOS Integrated System Architecture with Virtual Networks

notion of out-of-norm behavior, we introduce so-called out-of-norm detectors that are associated with the jobs. Finally, the section discusses the specification and execution of the out-of-norm detectors using timed automata.

## Out-of-Norm Behavior

Extending the notion of *behavior* from [C. 02], we denote a job's *behavior* as the *sequence of send and receive operations*. The behavior of a job is denoted as correct, if it is in accordance with the job's interface specification. Otherwise we speak of a job failure (i.e., a behavior violating the interface specification) and denote the respective job as faulty. In case of imprecise temporal interface specifications, however, such a demarcation between correct and faulty behavior proves to be difficult. While in a time-triggered communication system the precise temporal interface specification with a priori knowledge about the global points in time of message exchanges allows to definitely distinguish between correct and faulty temporal behavior, the imprecise interface specification of an event-triggered system complicates failure detection. When the temporal behavior of a job is determined by its inputs from the environment, an underlying model of the environment is necessary to evaluate correct job behaviors. For example, consider a user interface job in an automotive application that sends a message to a window lifter job whenever certain buttons are pressed. The ability to detect a faulty user interface job requires assumptions about the frequency and timing for pressing the button. Repeatedly sent messages within a short interval of time might represent a failure of the user interface job, or simply constitute an unanticipated customer behavior (e.g., playing with the window lifter button).

Although an omniscient observer can always demarcate between correct and faulty behavior, from within the system the available redundancy and a priori knowledge constrain the ability for performing a definitive classification. In order to handle imprecise temporal interface specifica-



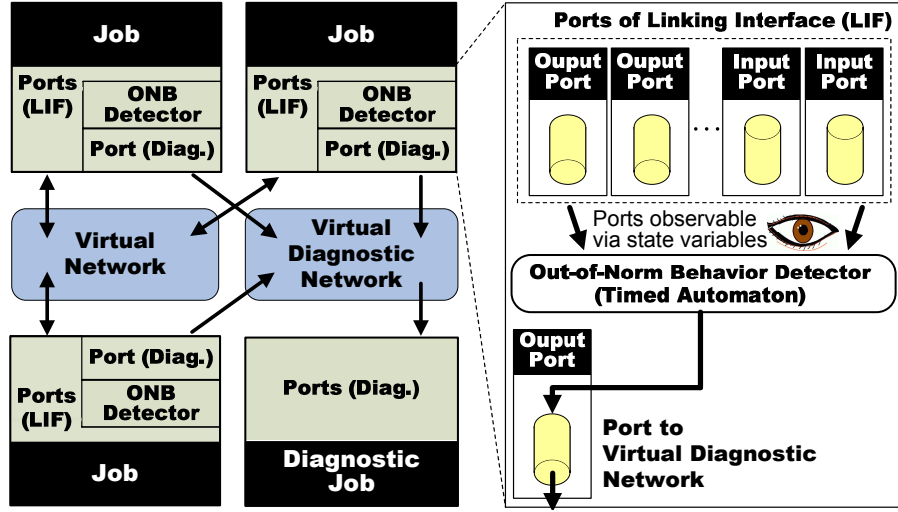


Figure 2. Out-of-norm Behavior (ONB) Detectors based on Timed Automata

tions with limited a priori knowledge, we introduce the notion of *out-of-norm behavior*. Behavior is denoted as out-of-norm, if it cannot be classified definitively as correct or faulty at the point in time of occurrence. Out-of-norm behavior represents an improbable behavior that probabilistically represents a failure.

The notion of *out-of-norm* originally comes from production machinery where early warnings of system breakdown enable preventative maintenance, thereby achieving enormous cost savings [JCGC97]. In this paper we extend this concept to the domain of computer systems by providing a generic mechanism that allows to detect suspicious behaviors in the temporal and value domain.

## Out-of-Norm Behavior Detector

At each job, a so-called *out-of-norm behavior* detector monitors the behavior at the ports to the virtual network. When out-of-norm behavior is detected, a corresponding diagnostic message is disseminated via a so-called *virtual diagnostic network*. This virtual diagnostic network is established by exploiting the high-level virtual network service. Such a virtual solution has two main advantages. At first, real-time traffic is not compromised in any way since the bandwidth for the exchange of diagnostic information is fixed a priori at design time. This way a deterministic message exchange for all non-diagnostic DASs is guaranteed. Secondly, the purely virtual solution ensures that no additional hardware faults are introduced due to wiring or connector problems. Consequently, *no probe effect* can be introduced [Gai86]. By exploiting the architectural high-level services to establish the virtual diagnostic network, no back-propagation of the diagnostic dissemination service to safety-critical subsystems is possible, since only elementary interfaces are used [Kop99].

Consequently, at least two ports are associated with every job. One port, which is an output port, is a connection to the virtual diagnostic network (see Figure 2). The other ports form the *linking interface* [KS02] of the job, via which the job is connected to the other jobs of the DAS. These latter ports are the access points to the DAS's virtual network, which serves the exchange of messages between jobs to realize the emergent services.

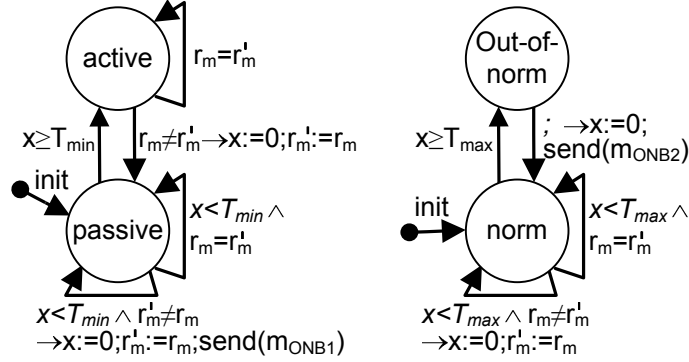


Figure 3. Example Automaton for Detection of Out-of-Norm Behavior

Whenever the out-of-norm behavior detector recognizes out-of-norm behavior at the ports of the linking interface, the detector sends a diagnostic message at the virtual diagnostic network with the following information:

- **Time:** The message sent to the virtual diagnostic network contains a timestamp denoting the point in time of the out-of-norm behavior detection.
- **Space:** The location (job and DAS) where the out-of-norm behavior has been detected.
- **Value:** The type of the out-of-norm behavior (e.g., with respect to the dynamics of the real-time entities, such as an improbable gradient in the message values) and the state variables that are significant for analyzing the out-of-norm behavior (i.e., contextual information).

### Specification of Out-of-Norm Behavior Detectors

Each out-of-norm behavior detector is specified using a deterministic timed automaton, i.e., a state transition graph annotated with timing constraints. We extend the timed automata defined in [AD94] by supporting not only clock variables, but also state variables with finite value domains. The state variables can be declared by the user within a timed automaton. In addition, predefined state variables are used to monitor the ports of the linking interface. These predefined state variables include:

- **State variables for messages.** A port to an event-triggered virtual network is a queue. Each position in this queue is made accessible to the out-of-norm behavior detector via a corresponding state variable. The access to this state variable does not affect the data at the port, i.e., no messages can be modified or retrieved from the queue.
- **State variables for port status.** These state variables contain information about the ports, such as the number of queued messages, and the total number of sent or received messages.

Figure 3 contains exemplary timed automata for monitoring the interarrival time of incoming messages. The automaton on the left-hand side of Figure 3 contains two states (active and passive) for detecting violations of the minimum interarrival time  $T_{\min}$ . The actual interarrival time of the message is captured with the clock variable  $x$ . The detection of the arrival of a message occurs using the predefined state variable  $r_m$  denoting the number of received messages.  $r_m$  is compared to a user-defined variable  $r'_m$  with the previous value of  $r_m$  in order to react to an increase of

the number of received messages. In the active state,  $x$  is larger than  $T_{\min}$  and a new message may arrive. In the passive state, the minimum interarrival time has not yet elapsed and the arrival of a message represents an out-of-norm behavior. In this case, a diagnostic message is sent ( $send(m_{ONB1})$ ) via the virtual diagnostic network. The information concerning the spatial and temporal context is automatically added.

The timed automaton on the right-hand side of Figure 3 detects violations of the maximum message interarrival time  $T_{\max}$ . This example is of particular importance for today's automotive applications, because many automotive ECUs provide a life sign with a maximum message interarrival time. In case the application in an ECU does not send a message with user data during an interval of length  $T_{\max}$  (e.g., park assist above 30km/h), a life sign message is produced controlled by a timeout. Like in the previous example, the clock variable  $x$  captures the actual interarrival time of the message.  $x$  is reset whenever a message arrives. In case  $x$  exceeds  $T_{\max}$ , an out-of-norm behavior has occurred and a diagnostic message is sent ( $send(m_{ONB2})$ ).

## Execution of Out-of-Norm Behavior Detectors

The execution of the timed automata is controlled by the progression of time on a global sparse time base [Kop92]. In the sparse time model the continuum of time is partitioned into an infinite sequence of alternating durations of activity and silence. Thereby, the occurrence of significant events is restricted to the activity intervals of a globally synchronized action lattice. The sparse time base allows to generate a consistent temporal order on the basis of time-stamps [Kop97]. During the silence intervals of the action lattice the sparse time base provides a consistent distributed state, where the notation of state is used as introduced in system theory [MT89, p. 45] as a dividing line between past and future.

Every port of a job is associated with a corresponding port state, which is the state of the job as seen from the port. The notation of *port state* is based on the concept of interface state [C. 02]. For a port with state semantics, the port state comprises a state variable, while a message queue is the port state of a port with event semantics. The update of the port state through the virtual network service occurs during the activity intervals of the global sparse time base. A message reception from a virtual network results in the update the state variable associated with a input port with state semantics or the insertion of an event message into a message queue of an input port with event semantics. At an output port with event semantics, message transmissions via a virtual network results in the removal of messages from an outgoing message queue. In the silence intervals of the sparse time base, globally consistent input is provided via the port state to the diagnostic services (see Figure 4).

In these silence intervals of the virtual network service, the out-of-norm behavior detector accesses the ports via the execution of the deterministic timed automata. Thereby, each automaton can start its execution with a globally consistent port state. At each activation, the automata associated with the job are executed and the labels associated with the transitions of the automata cause the observation of ports and the production of diagnostic messages.

Upon each activation time progresses by  $n$  ticks, which is reflected by all clock variables maintained in the timed automata. For each automaton, the out-of-norm behavior detector proceeds with evaluating guards and performing state changes with edges for which all guards are being fulfilled. In case no edge can be taken with the current values of the variables (including clock variables), the execution of the respective automaton is finished for this activation cycle. During the execution of the automata, the reading of the port state is triggered through the execution of

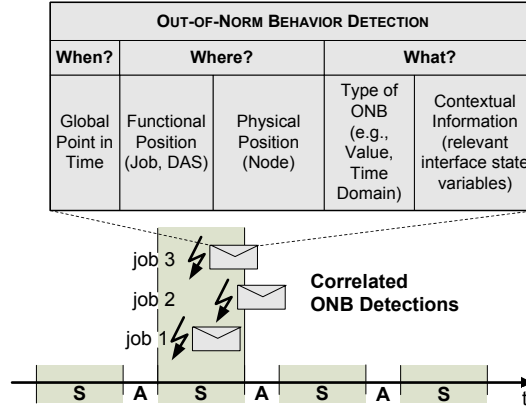


Figure 4. Diagnostic Messages. The architecture provides indications of out-of-norm behavior at the action lattice of a global sparse time base with corresponding activity (A) and silence (S) intervals.

transitions accessing the predefined state variables. The taking of an edge with a label *send(...)* allows to transmit a diagnostic message.

During the activity interval of the virtual network service, on the other hand, the out-of-norm behavior detector is inactive in order to prevent concurrent access of the port state between the timed automata and virtual network service.

#### 4. Exploitation of Out-of-Norm Behavior Detections

Once *out-of-norm behavior* of a job is detected, information about the occurrence of this behavior is disseminated so it can be used as input to other services at the architecture or application level:

- 1 **Maintenance services** process information about the occurrence of out-of-norm behavior to determine whether jobs or nodes need to be replaced.
- 2 **Engineering feedback services** process information about out-of-norm behavior in order to generate engineering feedback (e.g., performance monitoring, information regarding the bandwidth utilization).
- 3 **Membership services** are architectural services that solve the membership problem [Cri91] by achieving agreement on the identity of all correctly functioning jobs of a DAS. A membership service simplifies the provision of many application algorithms and plays an important role for controlling application level fault-tolerance mechanisms.

The key element to all above mentioned applications is the inclusion of meaningful information about the *space*, *time*, and *value* of an occurrence of out-of-norm behavior indicated through a diagnostic message. The information in a diagnostic message is trustworthy in the sense that this information has been acquired from independent instances, i.e., from architectural services executes at different jobs and at different nodes. Based on the fault hypothesis of the DECOS architecture [OPHS06], nodes are independent FCRs with respect to hardware faults and jobs are independent FCRs for software faults. We denote the underlying design concept as *cross checking* [POK05, Pet05], i.e., different FCRs are evaluating checks upon each other.

In the context of *space*, error containment between virtual networks [OPK04] ensures that the detected out-of-norm behavior enables to constrain the problem to a certain DAS. Consequently,

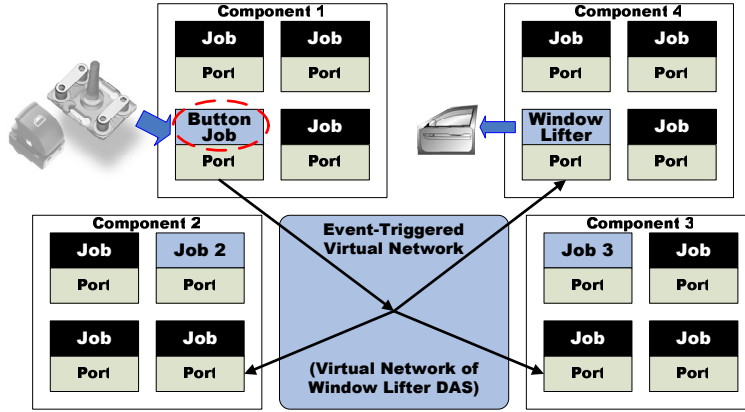


Figure 5. Exploitation of Out-of-Norm Behavior Detections

the out-of-norm behavior cannot result from interference with another DAS). Consider, on the other hand, a CAN system [Bos91] with two DASs. If the two DASs share a common CAN bus, then a failure in one DASs (e.g., babbling idiot failure with high priority messages) can propagate into the other DAS since the message transmissions of one DAS can delay message transmission of the other DAS,

Furthermore, the DECOS architecture guarantees that the predefined temporal properties for the communication activities hold also in case of job failures within the DAS [OPK04]. Although an out-of-norm behavior may propagate within a DAS, the out-of-norm behavior detection services allow to trace the job representing the origin of this phenomenon as well as the spreading within the DAS. Thus, error containment between jobs permits to pinpoint the jobs that are responsible for a specific out-of-norm behavior.

*Time* information is meaningful due to the availability of a global notion of time in the integrated DECOS architecture. The out-of-norm behavior detection events at different jobs occur within the same silence interval of the global sparse time base. Furthermore, all jobs in a DAS have access to a consistent port state, which is a prerequisite for the correlation of out-of-norm behavior detections resulting from the cross checking between jobs.

*Value* information captures the type of out-of-norm behavior (e.g., improbable frequency of messages sent by a job, implausible gradient of a sensor value) as well as contextual information given by the relevant part of the port state at the point in time of the out-of-norm behavior event.

Figure 5 exemplifies the exploitation of the out-of-norm behavior detection in an automotive example. Whenever a window lifter button is pressed, a corresponding job of the window lifter DAS disseminates a message with information about this event to an actuator job that accesses the motor. Out-of-norm behavior detectors are located at all jobs of the DAS. These jobs execute deterministic timed automata accessing the port state in order to determine a possible out-of-norm behavior of the button job.

In case the button job is subject of out-of-norm behavior induced by either a software or hardware fault, the other jobs of the DAS (actuator job, and jobs 2 and 3) will collectively detected this out-of-norm behavior. The checks are being executed in the other nodes in order to realize the cross checking principle and this information is forwarded to an analysis process. The analysis process exploits this information for the construction of maintenance services, engineering feedback services, or membership services. This analysis process concentrates this information

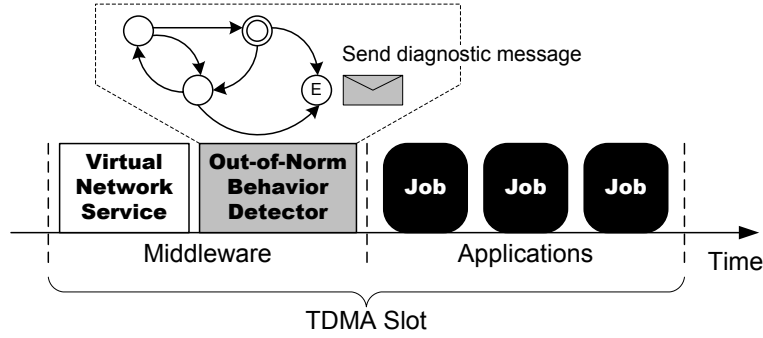


Figure 6. Overview on the Software Executed in each Node

and correlates it with other out-of-norm behavior detections from different functional elements (DASs, jobs) and information from different physical elements (nodes). For maintenance purposes, correlated job failures of different DASs within a node are an indication of a hardware fault [OP06]. Isolated job failures are an indication for software faults. The trustworthiness of the space, time, and value information in diagnostic messages is essential for determining during the analysis process whether a hardware or software faults is affecting the state of the system.

## 5. Implementation

The target platform used for the implementation of the introduced concept is based on the Time-Triggered Architecture (TTA) and exploits TTP [KG94] as the core physical network of the integrated system. The TTA provides a computing infrastructure for the design and implementation of dependable distributed embedded systems up to the highest criticality class [KB03].

The nodes employed in the prototype implementation use an embedded real-time Linux variant extended by a time-triggered scheduler as their operating system [HPOS05, Pet05]. For the communication system, the prototype uses an implementation of multiple virtual networks on top of TTP [OP05]. Each TTP frame is partitioned according to the number and requirements of the jobs of the DASs hosted on the respective DECOS node. This splitting of available communication resources and the establishment of the protocol specific properties (e.g., transformation of state variables into queues as required in event-triggered communication) is performed by the middleware deployed on each DECOS node.

For the implementation of the out-of-norm behavior detection service, we have added another middleware layer in this prototype implementation of the DECOS architecture. A generic out-of-norm behavior detector is parameterized by the operational interface specifications of jobs and executed as a time-triggered task. Each job's operational interface specification comprises an XML description as defined in [OP05b]. The XML description denotes the syntax of exchanged messages and one or more timed automata expressing different classes of out-of-norm behaviors for jobs. Out of this XML description, tools create software modules for the middleware task performing out-of-norm behavior detection. The out-of-norm behavior detector accesses the virtual network service, which is also realized as a middleware task, in order to acquire information about the job's message transmissions and receptions. This information controls the execution of the timed automata and therefore determines when diagnostic messages are sent via the virtual diagnostic network.

Figure 6 illustrates the software that is executed during each TDMA slot of a node. The execution of the virtual network service ensures that the messages for each job hosted on the node are transferred using the underlying TTP network. In the out-of-norm behavior detector, the timed automata are executed in order to detect improbable behaviors and deviations in the value and time domain from the respective port specification. Upon a detection event, a diagnostic message is constructed and disseminated via the virtual diagnostic network to the analysis subsystem for further assessment in order to determine the fault. As indicated by the time line in Figure 6, as soon as the middleware (i.e., for out-of-norm behavior detector and virtual network service) has terminated, the jobs can be executed (depending on the node configuration, one or more jobs are executed in one TDMA slot).

## 6. Discussion

The key for effective error detection is a precise job interface specification in the syntactic, temporal, and semantic domain. However, such sharp specifications may not always be available, especially in systems where flexibility is more important than predictability. Detection of out-of-norm behavior is especially useful, if the specification includes imprecise specifications such as probabilistic timing assumptions as in event-triggered communication systems. Typically, this information is expressed via probability distributions, thus a sharp line that allows a classification into correct and incorrect cannot be drawn. In this paper we provide a mechanism that allows to cope with this uncertainty by classifying this gray area as out-of-norm. The execution of timed automata that encode checks in the value and temporal domain allows to detect those interface states that bear the potential of revealing job faults.

Based on the error containment of the DECOS architecture provided via the underlying time-triggered core network and high-level encapsulation services, it can be assured that the detected out-of-norm behavior does not originate from unintended side effects or mutual dependencies between jobs having no functional dependencies. This ensures that out-of-norm behavior can be indisputably assigned to one or more jobs having functional dependencies. Furthermore, failure modes such as babbling idiot or masquerading failures are precluded by the architecture (i.e., virtual network service) and must not be dealt with at the application level.

While typically, executable assertions can only capture incorrect data in the value domain [MN88] the proposed solution can also capture out-of-norm behavior in the time domain, thus providing information about the dynamics of the system.

The introduced framework provides a solution to gather diagnostic information that can be used for either maintenance, feedback to jobs, or performance monitoring in order to judge about the effectiveness of the application functionality in the field. In case of maintenance, the information can be used as an input for online diagnosis in order to judge about the health status of the integrated system.

## Acknowledgments

This work has been supported in part by the European IST project ARTIST2 under project No. IST-004527 and the European IST project DECOS under project No. IST-511764.

## References

- [AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

- [Ade03] A. Ademaj. *Assessment of Error Detection Mechanisms of the Time-Triggered Architecture Using Fault Injection*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2003.
- [Bos91] Robert Bosch GmbH, Stuttgart, Germany. *CAN Specification, Version 2.0*, 1991.
- [C. 02] C. Jones et al. Final version of the DSoS conceptual model. *DSoS Project (IST-1999-11585)*, December 2002.
- [Cri91] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.
- [Fle05] FlexRay Consortium. BMW AG, DaimlerChrysler AG, General Motors Corporation, Freescale GmbH, Philips GmbH, Robert Bosch GmbH, and Volkswagen AG. *FlexRay Communications System Protocol Specification Version 2.1*, May 2005.
- [Gai86] J. Gait. A probe effect in concurrent programs. *Software Practice and Experience*, 16(3):225–233, March 1986.
- [HPOS05] B. Huber, P. Peti, R. Obermaisser, and C. El Salloum. Using RTAI/LXRT for partitioning in a prototype implementation of the DECOS architecture. In *Proc. of the Third Int. Workshop on Intelligent Solutions in Embedded Systems*, May 2005.
- [JCGC97] G.T. Jermy, R.C. Castle, C.F. Gimblett, and R. Chakrabarti. Monitoring out of normal conditions in repetitive cycle production machinery. In *Proc. of the Fifth Int. Conference on Factory 2000*, pages 29–33. IEEE, April 1997.
- [KB03] H. Kopetz and G. Bauer. The time-triggered architecture. *IEEE Special Issue on Modeling and Design of Embedded Software*, January 2003.
- [KG94] H. Kopetz and G. Grunsteidl. TTP – a protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, January 1994. Vienna University of Technology, Real-Time Systems Group.
- [Kop92] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proc. of 12th Int. Conference on Distributed Computing Systems*, Japan, June 1992.
- [Kop97] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.
- [Kop99] H. Kopetz. Elementary versus composite interfaces in distributed real-time systems. *ISADS '99, March 1999, Tokyo, Japan*, Mar. 1999.
- [KS02] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. Research report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.
- [MN88] B.M. McMillin and L.M. Ni. Executable assertion development for the distributed parallel environment. In *Proc. of the Twelfth Int. Computer Software and Applications Conference (COMPSAC 88)*, pages 284–291. IEEE, October 1988.
- [MT89] M. D. Mesarovic and Y. Takahara. *Abstract Systems Theory*, chapter 3. Springer-Verlag, 1989.
- [Obe04] R. Obermaisser. *Event-Triggered and Time-Triggered Control Paradigms – An Integrated Architecture*. Real-Time Systems Series. Kluwer Academic Publishers, November 2004.
- [OP05a] R. Obermaisser and P. Peti. Realization of virtual networks in the DECOS integrated architecture. In *Proc. of the Workshop on Parallel and Distributed Real-Time Systems 2006 (WPDRTS)*. IEEE, April 2005.
- [OP05b] R. Obermaisser and P. Peti. Specification and execution of gateways in integrated architectures. In *Proc. of the 10th IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA)*, Catania, Italy, September 2005. IEEE.
- [OP06] R. Obermaisser and P. Peti. A fault hypothesis for integrated architectures. In *Proc. of the 4th Int. Workshop on Intelligent Solutions in Embedded Systems*, June 2006.
- [OPHS06] R. Obermaisser, P. Peti, B. Huber, and C. El Salloum. DECOS: An integrated time-triggered architecture. *e&i journal (journal of the Austrian professional institution for electrical and information engineering)*, 3:83–95, March 2006. Available at <http://www.springerlink.com>.
- [OPK04] R. Obermaisser, P. Peti, and H. Kopetz. Virtual networks in an integrated time-triggered architecture. Research report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.



- [Pet05] P. Peti. *Diagnosis and Maintenance in an Integrated Time-Triggered Architecture*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, September 2005.
- [POK05] P. Peti, R. Obermaisser, and H. Kopetz. Out-of-norm assertions. In *Proc. of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, March 2005.
- [Rus01] J. Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, September 2001.



# A MODEL-DRIVEN FRAMEWORK FOR THE GENERATION OF GATEWAYS IN DISTRIBUTED REAL-TIME SYSTEMS

*Proceedings of the 28th IEEE Real-Time Systems Symposium, pages 93–104, Tucson, Arizona, USA, December 2007.*

Roman Obermaisser  
Vienna University of Technology  
Real-Time Systems Group  
romano@vmars.tuwien.ac.at

**Abstract** The DECOS integrated architecture supports the coexistence of multiple application subsystems (e.g., comfort, multimedia or braking subsystem in a car) on a single distributed computer system, thus significantly improving the utilization of hardware resources. In order to enable a component-based realization of such an integrated system with correctness-by-construction attributes (e.g., temporal composability), each application subsystem is assigned an encapsulated overlay network on top of a time-triggered physical network. As part of the DECOS architecture, this paper presents a generic framework for gateways, which enable message exchanges across application subsystem boundaries in order to exploit redundancy and to coordinate the behavior of application subsystems. A gateway can connect an overlay network to the overlay networks of other application subsystems, as well as to physical networks outside the integrated computer system (e.g., a fieldbus or a legacy cluster). In the DECOS architecture, not only physical networks, but also overlay networks of different application subsystem can exhibit property mismatches, such as different protocols (e.g., CAN overlay network vs. time-triggered overlay network), divergent syntax, and incoherent naming. Gateways provide a systematic solution for resolving these property mismatches. Within a gateway, a real-time database separates the application subsystems and stores temporally accurate real-time images. Controlled by a formal gateway specification based on an extension of timed automata, a gateway acquires messages from one gateway side to update these real-time images and recombines the real-time images into messages for the other gateway sides. In a prototype implementation, development tools use such a formal gateway specification expressed as a UML model as an input and automatically generate a configuration module for the parameterization of a generic architectural gateway service to a specific application.

**Keywords:** Integrated Architecture, Gateway, Timed Automata, Model-Based Design, Real-Time System

## 1. Introduction

Large distributed embedded systems (e.g., complete on-board electronic system of a car) consist of numerous application subsystems, each providing a part of the overall application functionality. Designers follow a divide-and-conquer strategy in order to manage the system's complexity by structuring the overall functionality into nearly-independent subsystems [Sim96, chap. 8]. For example, in-vehicle electronics are usually grouped into several domains, including the safety-related powertrain and chassis domains, as well as the non-safety critical comfort and multimedia domains [LH02]. Each domain comprises a set of ECUs interconnected by a network (e.g., CAN bus [Bos91]).

However, the subdivision of the overall system usually does not lead to fully independent application subsystems. Interactions between application subsystems are required for the following reasons:

- *Improved quality-of-service.* The service of one application subsystem can often be improved through the coordination with other application subsystems. For example, in an automotive system the wheel-speed sensors from the factory installed Antilock Braking System (ABS) can be exploited to estimate the car's heading for the navigation system during periods of GPS unavailability [CG02].
- *Application services that span more than one application subsystem.* Not all application services emerge from a single application subsystem. For example, the passive safety mechanism (Pre-Safe) of the Mercedes S-class [Bir03] correlates information of existing car dynamics sensors in order to determine hazardous situations such as skidding, emergency braking, or avoidance maneuvers.
- *Redundant sensors.* If multiple application subsystems measure the same physical entity in the environment, sensor redundancy can be exploited to improve reliability or accuracy of sensors. Alternatively, redundant sensors can be eliminated to reduce cost. For example, in-vehicle temperature sensors are used in multiple domains for state estimation, climate control, and accuracy-improvement of temperature-dependent sensors.

Motivated by the requirement of information exchanges between application subsystems, this paper focuses on a *systematic approach* for the realization of gateways in integrated distributed embedded systems. As part of the DECOS architecture [OP05], we present a gateway framework with an accompanying model-driven design process for interconnecting the networks of different application subsystems.

The DECOS architecture uses one time-triggered physical network as a backbone for the establishment of multiple overlay networks, each of which is dedicated to a respective application subsystem. Furthermore, the DECOS architecture supports additional physical networks – not necessarily time-triggered ones – in order to access fieldbus networks and federated legacy clusters.

A gateway is an architectural element that serves the controlled information exchange between multiple of these networks, each of which can be either an overlay network or a physical network. The purpose of a gateway is the *selective redirection of information* in conjunction with the necessary *property transformations*. Each of the overlay and physical networks runs a respective communication protocol (e.g., time-triggered exchanges of state messages [KG94, Fle05], CAN [Bos91], TCP/IP, etc.) and can exhibit its own namespace and message syntax.

The design process in the presented gateway framework starts with a *gateway specification* that formally describes the gateway. The formal gateway specification defines a *real-time database*, which is contained in the gateway and stores real-time images for the information exchange between the interconnected networks. In addition, the formal gateway specification defines the protocols to feed data into the real-time database, as well as the protocols to make accessible the data from the real-time database via the networks. For this purpose, the formal gateway specification is expressed with timed automata [AD94] extended with gateway-specific operations (e.g., operations for accessing the real-time database). The formal gateway specification serves as the input for an automatic code generation tool, which yields data structures and code that serve as a parameterization of a generic architectural gateway service.

In developing the conceptual foundations of this gateway framework and presenting a realization as part of an implementation of the DECOS architecture, the paper provides the following main contributions:

**Novel solution for gateways based on a real-time database.** Central to the introduced gateways is the notion of a real-time database as the interface between interconnected networks. Each network is associated with a timed automaton that specifies a protocol to either provide access to the real-time database or update it with information from the network.

**Modular construction of gateways.** The presented gateways follow a divide-and-conquer strategy, in which each timed automaton is a building block in the design of a gateway. Separated by the real-time database, timed automata for different types of networks can be independently developed and reused.

**Execution semantics and automatic code generation.** The formal gateway specification possesses execution semantics for time-triggered architectures (e.g., FlexRay [Fle05], TTP [KG94]). In a prototype implementation, tools take the gateway specification, which is constrained by a UML meta-model, as an input for the automatic generation of gateway code. Due to the reduction of software implementation faults and the speed up of development, such automatic code generation tools are becoming increasingly accepted in industry [WP99].

**Support for real-time systems encompassing multiple application subsystems.** The introduced gateways provide a solution for the construction of real-time systems with multiple application subsystems. Such a system will require the redirection of real-time images with temporal guarantees between the networks of different application subsystems.

**Legacy integration.** The presented gateway specification and tools support legacy networks with pre-existing protocols, message formats, and message names.

The paper is structured as follows. Section 2 presents related work on gateways. Section 3 contains an overview of the DECOS architecture, which is the time-triggered integrated architecture that encompasses the gateway framework presented in this paper. The explanation of this gateway framework, which offers generic architectural services that can be parameterized to specific applications, is the focus of Section 4. The parameterization occurs within a model-driven development process that uses a formal gateway specification to formally capture the information to control the behavior of the gateway (i.e., selective redirection and property transformations). Section 5 describes the structure of this formal gateway specification, which is constrained by a corresponding UML meta-model. The focus of Section 6 is an automatic code generator producing from the formal gateway specification a set of executables for a prototype implementation of the DECOS architecture. The entire development process is exemplified using an automotive scenario in Section 7.

## 2. Related Work

According to [Dod01], a gateway is *a network participant that provides automated interfaces to another network or system using different message formats and/or communications protocols*. Numerous gateways have been developed for the interconnection of specific communication protocols. In addition, work on gateway frameworks has explored generic solutions, which support

the generation of gateways based on formal specifications. Since the gateway framework presented in this paper employs state transition models that are linked by a real-time database, we will give an overview about related work on specifications for gateways based on state transition models. In addition, we will discuss solutions for the coupling of networks using a database with information shared between the respective application subsystems. Finally, gateway frameworks with support for specific transport abstractions will be investigated, because the gateway framework presented in this paper also employs predefined transport abstractions (i.e., state and event messages).

### Specification of Gateways using State Transition Models

Numerous protocol converters have been devised using state transition models with synchronous send or receive operations, thereby leading to Communicating Sequential Processes (CSP)-style rendezvous interactions between different protocol state machines. For example, [KLNS91] describes the generation of a gateway by computing the largest common subset of the services provided by two protocols. Protocols are modeled by Communicating Finite State Machines (CFSMs). [RM91] is a further example for protocol conversion using CFSMs. The authors present an algorithm that searches for legal traces in the cross-product of the state space of two protocols. Legal traces need to satisfy constraints that are captured by the so-called *converter specification*. The identified legal traces can be combined to a CFSM for the protocol converter.

### Coupling of Networks with a Database

The proxy architecture described in [CFMB98] supports the interconnection of two heterogeneous networks with a *reliable multicast proxy*. The proxy contains an agent for each of the two networks in order to maintain a database called the *data store*. In addition, the architecture proposes a protocol adapter to realize congestion control, data transformations (e.g., recoding of information with different data types), and protocol conversions.

Another example of a solution using a database for the interconnection of networks is the distributed object-oriented real-time database system called BeeHive [SSN98]. Data is incorporated into the database, if the data satisfies the real-time, fault tolerance, quality of service, and security characteristics that the user specified.

A third example for the network interconnection using a database is an architecture for avionic systems-of-systems based on real-time publish/subscribe and High-Level Architecture (HLA) [FPT05]. The architecture proposes a database interface and four roles of software modules. Input modules write data into the database. Logic modules perform computations on the contents of the database. Output models read the results of the logic modules from the database. The architecture introduces a central manager that is part of the system software and provides access to the database for the input, logic, and output modules.

### Gateways with Predefined Transport Abstractions

Based on the services and features of transport protocols [IAC99], different types of transport abstractions can be distinguished. For example, the protocol conversion toolkit [Aue89] introduces a set of four transport abstractions, namely datagrams, streams, sequenced packets, and conversation abstraction types. In case different transport abstraction need to be connected, a transport abstraction converter resolves the resulting property mismatch.

## Relationship to Presented Work

The gateways presented in this work couple different networks using a real-time database. In contrast to other database-centric solutions, such as the proxy architecture in [CFMB98], we offer a model-driven development process and focus on real-time systems that require guaranteed temporal properties (e.g., bounded latency in the redirection of messages). The gateway maintains the temporal accuracy of the real-time images in the database in order to ensure that real-time images have not been invalidated by the progression of time when they are used. Timed automata, which are extended by communication actions, control the access of the real-time images in the database, as well as the transmission and reception of messages at the interconnected networks. Each network is equipped with a corresponding timed automaton, which encodes the protocol of the respective network. The reason for the use of timed automata [AD94] in contrast to the formal specifications in Section 2 is the ability of expressing guards and actions w.r.t. to physical time.

In analogy to the protocol conversion toolkit [Aue89], the presented gateways also support distinct transport abstractions. The presented gateways support communication primitives for *event messages* and *state messages*, which have been identified as two important message classes in distributed real-time systems [Kop97]. This bivalent distinction of event and state messages can also be found in IMA, where the two message types are referred to as sampled and queued mode [Aer06].

## 3. The DECOS Integrated Architecture

The DECOS architecture [OP05] offers a framework for the design of large embedded real-time systems by physically integrating multiple application subsystems on a single distributed computer system. The DECOS architecture offers to system designers generic architectural services, which provide a validated stable baseline for the development of applications. After presenting the conceptual model for structuring a DECOS system at the logical and physical level, this section will describe two of these architectural services, namely the *virtual network service* and the *gateway service*.

### Structuring of a DECOS System

A DECOS system (e.g., the complete on-board electronic system of a car) provides to its users (e.g., human operator) application services at the controlled object interface. From a logical point of view, a DECOS system consists of a set of nearly-independent *DASs*. Each DAS provides a subset of the overall application services, which is meaningful in the application context (e.g., steer-by-wire subsystem in the automotive domain). In analogy to the structuring of the overall system, we further decompose each DAS into smaller units called *jobs*. A *job* is the basic unit of work and employs a *virtual network* in order to exchange messages with other jobs and work towards a common goal.

From a physical point of view, a DECOS system encompasses a cluster containing a set of *node computers* (nodes for short), which are interconnected by a time-triggered physical network. The virtual networks are implemented on top of this time-triggered physical network. The use of a time-triggered physical network matches the predictability and fault-tolerance requirements of safety-critical applications [Rus01].

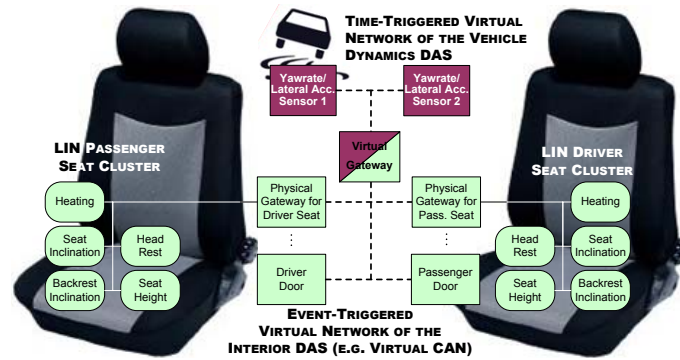


Figure 1. Exemplary Virtual and Physical Gateways

## Virtual Network Service

In the DECOS architecture, each DAS is provided with a dedicated communication infrastructure that is realized as a *virtual network*. A virtual network is established as an overlay network on top of a time-triggered physical network [OP05].

A job can send two types of messages on a virtual network, namely *state messages* and *event messages*. These two message types differ w.r.t. the information semantics [Kop97, p. 31]. While state messages contain the absolute value of a real-time entity (e.g., speed is  $10 \text{ ms}^{-1}$ ), event messages transport relative values (e.g., decrease of speed by  $2 \text{ ms}^{-1}$ ). The message type determines the interface (i.e., the ports) between the job and the virtual network.

A *state port* supports the reception or transmission of state messages. Since applications are often only interested in the most recent value of a real-time entity, a state port contains a memory element that is overwritten with newer state values whenever a message arrives at the port (i.e., update-in-place).

An *event port* supports the reception or transmission of event messages. In order to reconstruct the current state of a real-time entity from messages with event semantics, messages are queued at an event port in order to process every message exactly-once. The loss of a single message with event semantics could affect state synchronization between a sender and a receiver.

Based on state and event ports, virtual networks with higher protocols have been established on top of a time-triggered physical network. Examples are CAN [Nex03], a transport protocol for a hard-real time CORBA broker [SLO03], and TCP/IP [Nex03].

## Virtual and Physical Gateways

The need for gateways in the DECOS architecture arises either through the logical or physical system structuring. In case the coordination of the behavior of two DASs requires an information exchange between them, the logical structuring of the system into DASs induces so-called *virtual gateways*. The purpose of introducing this new architectural element instead of directly connecting all jobs through a single virtual network is twofold:

- 1 **Encapsulation of virtual networks.** A particular message is only redirected from one virtual network to a second one, if the virtual gateway has been explicitly parameterized to forward this message. Hence, jobs perceive only those messages from another DAS, which are required for realizing the DAS's application service. This strategy minimizes the mental effort for understanding a DAS and its constituting jobs, because the designer can



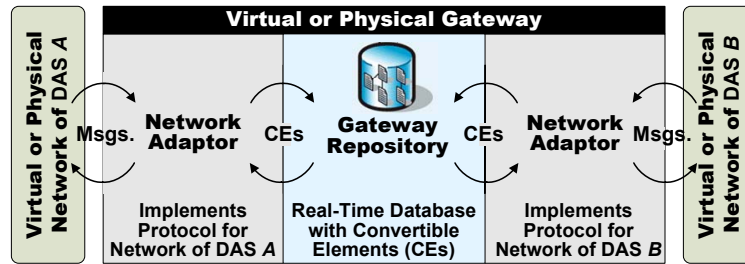


Figure 2. Gateway

abstract from all messages in other DASs that are not explicitly imported through a virtual gateway. In addition, the provision of a separate encapsulated virtual network for each DAS establishes clear error propagation boundaries.

- 2 **Resolving of property mismatches.** Since each virtual network is an overlay network that can have its own communication protocol and name space, virtual networks cannot be directly interconnected. The coupling of virtual networks requires a mediating entity – the virtual gateway – that resolves the property mismatches.

In addition to virtual gateways, the physical structuring of the overall system into a set of clusters, each with a separate physical network, induces so-called *physical gateways*. A physical gateway is realized by a node with physical connections to two physical networks. A physical gateway is needed for the information exchange between jobs, if the jobs of a DAS are allocated to more than one cluster.

Figure 1 shows an automotive example employing both virtual and physical gateways. The interior DAS in this example comprises the body electronics of the passenger compartment. As its communication infrastructure, this DAS uses an event-triggered virtual network, which is connected via physical gateways to the field bus networks embedded in the seats. In addition, the exemplary automotive system employs a virtual gateway for coupling the interior DAS with the vehicle dynamics DAS. The virtual gateway between the vehicle dynamics DAS and the interior DAS realizes pre-crash functionality, e.g., by tensioning seat-belts and realigning seats as described in [Bir03].

#### 4. Gateways based on a Real-Time Database

A real-time system can be modeled using a set of *real-time entities* [KK90], which are significant state variables that are located in the computer system or the environment. The current picture of such a real-time entity is called a *real-time image* and can be sent within a message on a virtual or physical network. Redirection of information through a gateway occurs when a real-time image contained in a message is required by another DAS connected to the gateway. We denote such a real-time image that is relevant at the gateway as a *convertible element*.

The presented gateways recombine convertible elements acquired from one network into messages for another network, while converting between different temporal and syntactic specifications and resolving naming incoherences. For this purpose, the gateway maintains a real-time database with convertible elements called the *gateway repository*. The gateway repository decouples the different networks accessed by the gateway and allows the convertible elements that are necessary for constructing a particular message to arrive at different points in time.





Information Semantics	Convertible Element Data	Meta Information	
		Variable	Description
Convertible Element 1 with State Semantics		$t_{update}^1$ $d_{offset}^1$ $b_{req}^1$	most recent update instant temporal accuracy offset update request indication
Convertible Element 2 with State Semantics		$t_{update}^2$ $d_{offset}^2$ $b_{req}^2$	most recent update instant temporal accuracy offset update request indication
⋮			
Convertible Element $k$ with Event Semantics		$b_{req}^{k+1}$ $n^{k+1}$	update request indication number of queued instances
Convertible Element $k+1$ with Event Semantics		$b_{req}^{k+2}$ $n^{k+2}$	update request indication number of queued instances
⋮			

Figure 3. Gateway repository contains convertible elements.

In addition, the gateway contains for each accessed DAS a so-called *network-adaptor*, which implements the communication protocol of the virtual or physical network of the DAS and performs information exchanges between the network and the gateway repository (see Figure 2).

## Network Adaptors

A *network adaptor* interacts with either a virtual network or a physical network according to the respective protocol. Firstly, a network adaptor can acquire convertible elements from a network and write them into the gateway repository. Depending on the protocol, the acquisition of a message with convertible elements can involve the exchange of several messages at input and output ports, e.g., the transmission of a request message before a response message carrying the convertible elements arrives. Secondly, a network adaptor can read convertible elements from the gateway repository, construct messages and disseminate them on a network. Thereby, information can be redirected between virtual and physical networks, if the read convertible elements have been placed in the gateway repository by another network adaptor.

The specification of the network adaptors occurs using an extension of deterministic timed automata with corresponding execution semantics and will be explained in Section 5.

## Gateway Repository

For the storage of convertible elements, the gateway repository takes into account the information semantics of convertible elements (see Figure 3). Due to the respective characteristics of state and event semantics (cf. Subsection 3), the gateway repository distinguishes two types of storage elements in analogy to state and event ports. For convertible elements with state semantics, the repository contains state variables that are overwritten whenever a new version of the convertible element arrives (update-in-place). Convertible elements with event semantics, on the other hand, are stored in queues.

In addition to the data of the convertible elements, the gateway repository also stores meta-information about convertible elements. The meta-information maintained in the gateway repository includes the following four attributes:

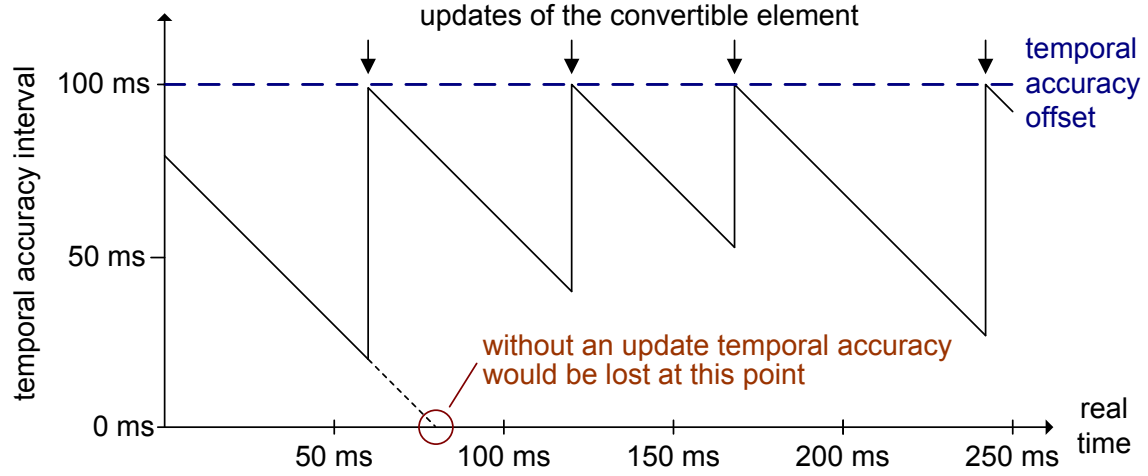


Figure 4. Temporal Accuracy

**Most recent update instant.** The point in time of the most recent update  $t_{\text{update}}$  is a dynamic attribute associated with each convertible element with state semantics.  $t_{\text{update}}$  is set to the current time  $t_{\text{now}}$ , whenever a network adaptor overwrites the convertible element in the gateway repository.

**Temporal accuracy interval and offset.** Due to the dynamics of real-time entities, which change their state as time progresses, the validity of convertible elements is time-dependent. Based on the notion of *temporal accuracy* [Kop97, p. 103], we can express constraints on the age of convertible elements. The age of a convertible element at its time of use introduces an error in the value domain, which is determined by the gradient of the real-time entity. Depending on the dynamics of a real-time entity and the maximum acceptable error in a given application, a gateway is allowed to store a convertible element only for a limited duration before the convertible element is invalidated by the progression of time.

For this reason, the gateway repository maintains for each convertible element with state semantics a dynamic attribute called the *temporal accuracy interval*  $d_{\text{acc}}$ . At any given instant,  $d_{\text{acc}}$  denotes how long the convertible element will still remain a valid image of the respective real-time entity in case no update of the convertible elements occurs in the meantime. Over time, the temporal accuracy interval conforms to an inverse sawtooth wave, in which the wave ramps downward and then sharply rises when an update of the convertible element occurs (see Figure 4).

The *temporal accuracy offset*  $d_{\text{offset}}$  is a static attribute that determines the temporal accuracy interval immediately after an update of the convertible element. In conjunction with the instant of the most recent update, the temporal accuracy offset allows to compute the temporal accuracy interval of a convertible element:

$$d_{\text{acc}} = d_{\text{offset}} - (t_{\text{update}} - t_{\text{now}}) \quad (1)$$

Hence, only the temporal accuracy offset needs to be stored in the gateway repository, because the temporal accuracy interval can be computed on-the-fly.

**Update request indication.** In order to support on-demand communication activities, the gateway repository contains boolean *update request indications*. For a convertible element with state

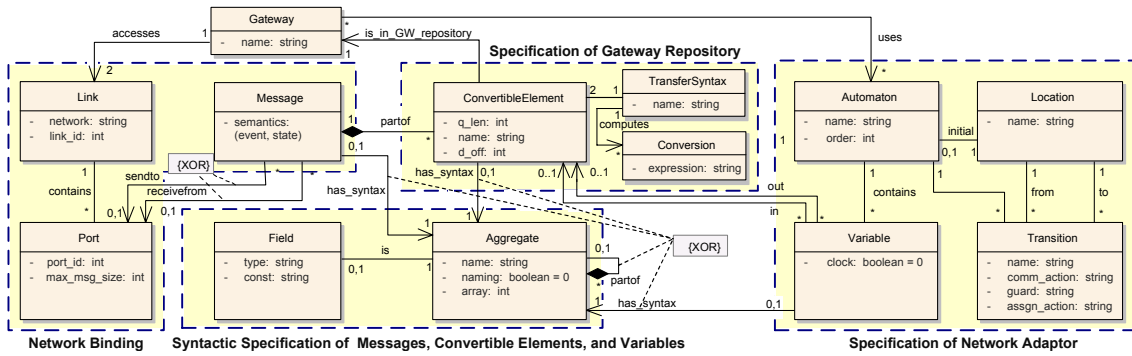


Figure 5. Gateway Specification Meta-Model

or event semantics, the respective update request indication  $b_{\text{req}}$  denotes whether a new convertible element needs to be transferred into the gateway repository. By setting the update request indication, a network adaptor can demand convertible elements from the other network adaptors. A network adaptor receiving messages from a network can initiate receptions conditionally, based on the value of the update request indication.

**Number of queued instances.** Every convertible element with event semantics possesses this dynamic attribute. It denotes the number of instances of the convertible element that are currently queued in the gateway repository.

Using the introduced attributes, we can control the behavior of a network adaptor as follows:

**Preserve temporal accuracy of convertible elements in gateway repository.** The meta-information provides the network adaptors with information for the decision whether to actively engage in the acquisition of convertible elements for the update of the gateway repository. A network adaptor can react to the imminent invalidation of temporal accuracy, e.g., by starting a protocol to perform an update of the convertible element in the gateway repository.

**Forward temporally accurate convertible elements only.** In conjunction with information concerning the transmission delay from the gateway to the receiver jobs, the gateway can decide to send messages only if all constituting convertible elements will be temporally accurate upon the reception by the receiver jobs.

**On-demand updating of convertible elements.** In order to efficiently use communication resources (e.g., bandwidth of a virtual network) and computational resources (e.g., CPU time at a sender job), the update of convertible elements in the gateway repository can occur conditionally. Using an *update request indication*, a network adaptor can express to the other networks adaptors that it requires a certain convertible element.

## 5. Gateway Specification Model

The gateway specification model is the starting point for the tool-supported development of gateways in the DECOS architecture. The gateway specification model enables developers to formally define the selective redirection of messages across DASs and the respective property

transformations on a high level of abstraction that hides the implementation details of the gateway services. A meta-model expressed in UML constrains the formulation of the gateway specification model and ensures that all information required for the instantiation of the generic gateway service is available. The constituting parts of the gateway specification model (see Figure 5) are the network binding, the syntactic specification, the specification of the gateway repository, and the specification of the network adaptors.

## Network Binding

This part of the gateway specification model defines the mapping of a gateway to the interconnected virtual and physical networks. The binding to a network comprises multiple Ports, which are grouped to a so-called Link. Each of the Links names the (virtual or physical) network that is accessed via the link (attribute *network*) and provides a unique link identification within that network (attribute *link\_id*). Ports are identified by a unique port identification within the Link (attribute *port\_id*) and possess a data direction. The gateway either sends Messages at a port (in case of an output port) or receives Messages from a port (in case of an input port).

## Syntactic Specification of Messages, Convertible Elements, and Variables

The syntactic specification (see UML class diagram in Figure 5) defines the structure of Messages, Convertible elements and Variables in terms of smaller structural elements denoted as Aggregates and Fields. An Aggregate possesses a unique name and allows the hierarchic grouping of structural elements. An Aggregate that is subdivided no further and considered atomic at the gateway is called a Field. A Field possesses a type (e.g., integer, string, floating point number). A Field is static, if the value of the field is time-invariant (value defined via the optional attribute *const*).

Variables are used to maintain an internal state in the timed automata comprising the specification of the network adaptors (cf. Section 6). In particular, Variables can store messages and convertible elements. They serve as the source for the transfer of a convertible element into the gateway repository or the transfer of a message to a port for being transmitted. In analogy, Variables serve as the destination when reading a convertible element from the gateway repository or receiving a message from a port.

A Variable, which contains only a single Field of integer type, can be marked as a clock variable (boolean attribute *clock*) in order to be autonomously incremented with the progression of the cluster's global time. The value of the clock variable denotes the number of ticks of the global time base since the epoch in a time-format standardized by the OMG [OMG02]. If a DECOS cluster contains multiple gateways distributed on separate nodes, all clock variables in a DECOS cluster are synchronously incremented. This synchrony of clock variables simplifies the coordination between gateways and the construction of replicated gateways. The basis for the synchrony is the global time base of the time-triggered physical network, which forms the core of the DECOS architecture and serves for the realization of the virtual networks (cf. Section 3).

In contrast to the Variables of the timed automata, each Message requires a *message name* that provides an identification by which the message is demarcated from other messages at a port. We can distinguish between two types of names:

**Explicit name.** The name is a subset of the constituting Aggregates of a message, which uniquely identifies the message when received at a port. The boolean attribute *naming* denotes

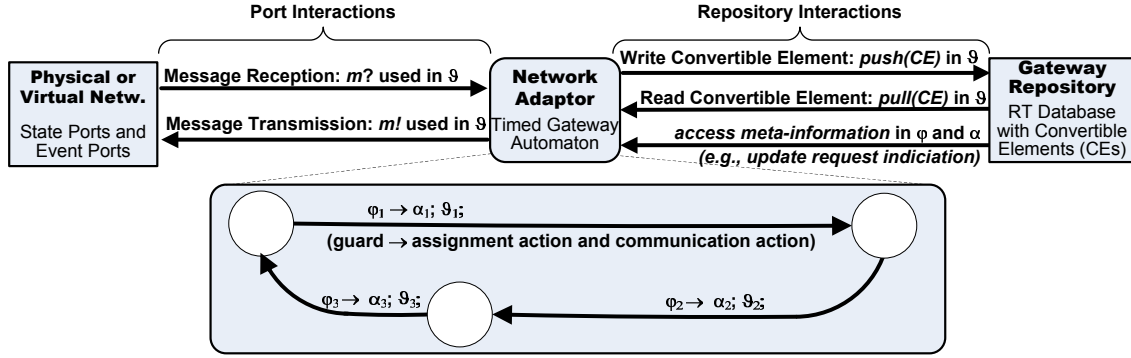


Figure 6. Port and Repository Interactions in Guards and Actions of a TGA

whether an Aggregate is part of the message name. In this case, all constituting Aggregates are also regarded as parts of the message name and all Fields must be static.

Consider for example a CAN message with standard frame format [Bos91]. The message name consists of the 11-bit identifier in conjunction with the Remote Transmission Request (RTR) flag. The message name serves both as an identification for the message contents of up to 8 data bytes, as well as for controlling the arbitration process.

**Implicit name.** If a port exclusively serves for the sending or receiving of a single message only, then the port implicitly identifies the message. The implicit naming via the port is mandatory for messages that do not contain an explicit name as part of the message contents. An example for such a message is a TTP frame [KG94], which is uniquely identified on the network by its TDMA slot, i.e., the global time of its transmission start instant.

## Definition of Gateway Repository

The gateway repository stores Convertible elements for the information exchange between the network adaptors. As described in Section 2, the updating and the storage of Convertible elements depends on the information semantics. Whether the Convertible element possesses state or event semantics is known via the attribute *semantics* of the associated Message. In case of event semantics, the attribute *q.len* denotes the maximum number of instances of a Convertible element that can be stored in the gateway repository. The attribute *d.off* is significant for Convertible elements with state semantics and denotes the temporal accuracy offset.

Furthermore, for each convertible element in the gateway repository a Transfer Syntax can be specified, which serves the automatic conversion between different syntactic representations of convertible elements. Each Transfer Syntax is associated with a triggering Convertible element and a Convertible element that shall store the result of the conversion. Upon a change of value of the triggering Convertible element, the expression of the Transfer Syntax is evaluated in order to compute a new value for the targeted Convertible element.

Transfer Syntaxes simplify the design of a gateway by automating syntactic conversions. In addition, a particular Transfer Syntax can be reused in different applications, e.g., by providing a library of commonly used conversions.

## Specification of Network Adaptors

The specification of the network adaptors is performed with deterministic timed automata [AD94], i.e., state transition graphs annotated with timing constraints. The employed timed automata, which are extended in order to support interactions with virtual and physical networks and the gateway repository, are called *Timed Gateway Automata (TGA)*. In the UML model in Figure 5, the classes Automaton, Transition and Location are used for defining the TGA. The class Automaton assigns a unique name to a TGA and provides an optional attribute *order* that allows to enforce a predefined execution order of multiple TGA (cf. Section 6). The Locations are the vertices of the graph and interconnected by edges called Transitions. The taking of a transition is instantaneous, whereas time can elapse within a location.

We use a notation based on guard and action labels (attributes *guard*, *assgn\_action*, and *comm\_action* of Transition). As depicted in Figure 6, guards and actions can encompass port interactions for the transmission and reception of messages. Furthermore, guards and actions can include repository interactions for reading and writing convertible elements in the gateway repository and accessing the meta-information.

Formally, a guard  $\varphi$  is specified via the following grammar:

$$\begin{aligned} \varphi := & \ x \circ v \mid x \circ c \mid v \circ c \mid v \circ v' \mid \text{avail}(m) \mid \text{avail}(c) \mid \\ & b_{\text{req}}(c) \mid d_{\text{acc}}(c) \circ z \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \end{aligned} \quad (2)$$

where  $x$  is a clock,  $z$  is a constant (in  $\mathbb{Z}$ ),  $v$  and  $v'$  are variables,  $m$  is a message,  $c$  is a convertible element, and  $\circ$  is a binary operator ( $\leq, <, =, >, \geq$ ).

Boolean expressions on variables and clocks are the basic elements of guards in a timed automaton [AD94]. In addition, the guard of a TGA can contain conditions on the temporal accuracy interval  $d_{\text{acc}}(c)$  and the update request indication  $b_{\text{req}}(c)$  of a convertible element  $c$  in the gateway repository. Further constituting elements of a guard are constraints concerning the availability of messages at ports or the availability of convertible elements in the gateway repository. A convertible element  $c$  is available (expressed as  $\text{avail}(c)$ ), if the temporal accuracy interval is positive (in case of state semantics) or the number of queued convertible element instances is positive (in case of event semantics):

$$\text{avail}(c) = \begin{cases} d_{\text{acc}}(c) > 0 & \text{if } c \text{ has state semantics} \\ n(c) > 0 & \text{if } c \text{ has event semantics} \end{cases}$$

For a message  $m$ , the availability constraint (expressed as  $\text{avail}(m)$ ) is always satisfied in case the message is located at a state port. A state port always stores a state message, since the reception at a state port is not consuming. For an event port, the queue in the event port must be not empty for the availability constraint to be satisfied. The information about the number of queued messages at an event port is provided by the virtual network services or the driver establishing the generic port-based interface on top of the physical network.

In addition to guards, transitions can be associated with *assignment actions* and *communication actions*. An assignment action  $\alpha$  adheres to the following grammar:

$$\alpha := x := z \mid v := z \mid v := v' \mid \text{req}(c) \mid \alpha_1; \alpha_2, \quad (3)$$

where  $x$  is a clock,  $z$  is a constant (in  $\mathbb{Z}$ ),  $v$  is a variable, and  $c$  is a convertible element. The operation  $\text{req}(c)$  causes the setting of the update request indication of the convertible element  $c$  in the gateway repository.

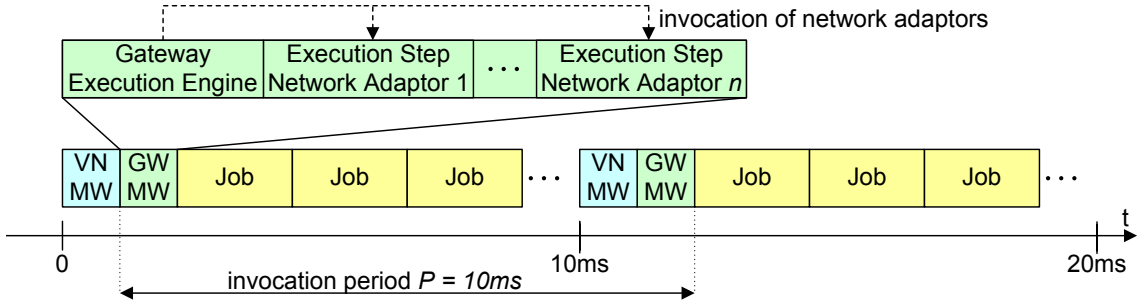


Figure 7. Time-Triggered Invocation of Virtual Network Middleware (VN MW) and Gateway Middleware (GW MW)

A communication action allows to access the gateway repository and the ports to a virtual or physical network. The grammar for the communication action  $\vartheta$  is:

$$\vartheta := \text{push}(c) \mid \text{pull}(c) \mid m! \mid m? \mid \vartheta_1 \wedge \vartheta_2 \quad (4)$$

The operation  $\text{push}(c)$  causes the transfer of a convertible element  $c$  from a variable of the TGA into the gateway repository. Inversely,  $\text{pull}(c)$  fetches a convertible element from the gateway repository and writes it into a TGA variable.

$m?$  performs the reception of message  $m$  when the transition is taken. The reception means that the message  $m$  is read from an input port and written to local variables of the TGA. The operation  $m!$  performs the transmission of message  $m$ . Prior to passing the message  $m$  to the output port, the current values of a specified set of the TGA's variables are used to set the contents of message  $m$ .

## 6. Implementation

The implementation of the DECOS cluster with virtual networks and gateways is an instantiation of the DECOS architecture introduced in Section 3. The DECOS cluster consists of five nodes and a physical network executing the TTP [KG94] for the interconnection of these nodes. The nodes host the application software (i.e., jobs belonging to one or more DASs) in conjunction with the DECOS architectural services (i.e., virtual networks and gateways). The nodes are implemented using Soekris net4521 embedded computers<sup>1</sup>. As the operating system, the nodes use a real-time Linux variant extended by a time-triggered scheduler [HPOS05]. Time-triggered tasks are used both for executing the jobs containing the application code, as well as for the middleware implementing the DECOS architectural services.

In the implementation, the middleware consists of two parts. The *virtual network middleware* realizes the virtual networks as overlay networks on top of the time-triggered transport service provided by the TTP controller in the nodes. The *gateway middleware* is responsible for the establishment of virtual and physical gateways. The virtual network middleware and the gateway middleware are periodically invoked by the time-triggered Linux-based scheduler, namely once in each TDMA round (10 ms in the prototype DECOS cluster). For example, in Figure 7 the virtual network middleware and gateway middleware have been assigned two time slots at the beginning of the TDMA round.

<sup>1</sup>[www.soekris.com](http://www.soekris.com)



In the following, the gateway middleware and its generation from the gateway specification model will be explained. A description of the implementation of the virtual network middleware can be found in [OP05].

## Gateway Execution Engine

Upon the invocation of the gateway middleware, control is passed to the *gateway execution engine*. The gateway execution engine, which is described via pseudo code in Figure 8, represents the core of the gateway middleware and incorporates the following functionality:

**Network binding.** The gateway execution engine provides to the network adaptors operations to send and receive messages at ports. For this reason, configuration data structures (Lines 1–3 in Figure 8) define the messages that are exchanged via state and event ports. The operation `send` in Line 24 accepts a message from a network adaptor and either enqueues this message at an event port or overwrites the current message of a state port. The operation `receive` in Line 26 stores a received message in a variable provided by the network adaptor. For this purpose, it dequeues a message at an event port or reads the most-recent message at a state port.

**Gateway repository.** The gateway execution engine contains declarations of the convertible elements that belong to the gateway repository. Depending on the information semantics a convertible element is either a single variable with updates-in-place for state semantics (Line 4) or a queue for event semantics (Line 5). Both types of convertible elements are stored in the gateway repository (Line 6).

Network adaptors can access the gateway repository using the operations `push` and `pull`. The operation `push` in Line 12ff accepts a variable from the network adaptor and stores its contents as a convertible element in the gateway repository. Inversely, `pull` (Line 21ff) retrieves a convertible element from the gateway repository and stores the contents in a variable specified by the network adaptor.

**Transfer syntax.** In Lines 7 to 9, the gateway execution engine is parameterized with the transfer syntax for convertible elements with state semantics and event semantics. Upon the change of a convertible element in the gateway repository through a `push` operation, the transfer syntax updates dependent convertible elements in Lines 16 and 19.

**Invocation of network adaptors.** After the invocation through the time-triggered RTAI/LXRT-based scheduler, the gateway execution engine on its behalf passes control to the network adaptors (Line 28). The invocation sequence of the network adaptors is controlled by the order of the entry points in the the array  $N_{\text{adaptor}}$ . This array is sorted by the order-attribute contained in the formal gateway specification (cf. Section 6). The gateway execution engine is reinvoked whenever one of the operations described above is activated or after the termination of the network adaptor.

## Network Adaptors

The gateway middleware can contain multiple network adaptors, each of which is specified with a corresponding TGA. Upon the periodic activation of the network adaptor within the gateway middleware, an execution step of the TGA takes place. The period  $P$  of the execution steps is equal to the activation period of the gateway middleware.

```

1   $m_{state} = \langle \text{data} : v, \text{meta} : \langle \rangle, \text{semantics} : state \rangle$ 
2   $m_{event} = \langle \text{data} : v[], \text{meta} : \langle n_{queued} \rangle, \text{semantics} : event \rangle$ 
3   $P = \langle id_{vn}, id_{port}, m_{state}[], m_{event}[] \rangle$ 
4   $c_{state} = \langle \text{data} : v, \text{meta} : \langle d_{offset}, b_{req}, t_{update} \rangle, \text{semantics} : state \rangle$ 
5   $c_{event} = \langle \text{data} : v[], \text{meta} : \langle b_{req}, n_{queued} \rangle, \text{semantics} : event \rangle$ 
6   $R = \langle c_{state}[], c_{event}[] \rangle$ 
7   $S = \langle s_{state}[], s_{event}[] \rangle$ 
8   $s_{state} = \langle c_{state}^{src\_ref}, c_{state}^{dst\_ref}[], \text{conversion} \rangle$ 
9   $s_{event} = \langle c_{event}^{src\_ref}, c_{event}^{dst\_ref}[], \text{conversion} \rangle$ 
10  $N_{adaptor} = \langle \text{entry\_point}[] \rangle$ 

11 // push operation : transfer contents of a TGA variable as a convertible elements into
12  $\text{push}(c^{ref}, v^{ref})$  : // the gateway repository and execute transfer syntax
13   if ( $c^{ref}.\text{semantics} = state$ )
14      $c^{ref} = v^{ref}, c^{ref}.\text{meta}.t_{update} = t_{now}$ 
15     if  $\exists s_{state} \in S$  with  $s_{state}.c_{state}^{src\_ref} = c^{ref}$ 
16        $\rightarrow \forall c_{event}^{dst\_ref} : c_{event}^{dst\_ref}.v = \text{conversion}(c_{event}^{src\_ref}), c_{event}^{dst\_ref}.\text{meta}.t_{update} = t_{now}$ 
17   else
18      $\text{enqueue}(c^{ref}.v, v^{ref})$ 
19     if  $\exists s_{state} \in S$  with  $s_{state}.c_{event}^{src\_ref} = c^{ref} \rightarrow \forall c_{event}^{dst\_ref} : \text{enqueue}(c_{event}^{dst\_ref}, \text{conversion}(c_{event}^{src\_ref}))$ 
20   end

21 // pull operation : retrieve conv. elements from repository and store in TGA variable
22  $\text{pull}(c^{ref}, v^{ref})$  : if ( $c^{ref}.\text{semantics} = state$ )  $v^{ref} = c^{ref}.v$  else  $v^{ref} = \text{dequeue}(c^{ref}.v)$ 

23 // send operation : send msg. at a port with the contents of a TGA variable
24  $\text{send}(m^{ref}, v^{ref})$  : if ( $m^{ref}.\text{semantics} = state$ )  $m^{ref}.v = v^{ref}$  else  $\text{enqueue}(m^{ref}.v, v^{ref})$ 

25 // receive operation : receive msg. at a port and store contents in a TGA variable
26  $\text{receive}(m^{ref}, v^{ref})$  : if ( $m^{ref}.\text{semantics} = state$ )  $v^{ref} = m^{ref}.v$  else  $v^{ref} = \text{dequeue}(m^{ref}.v)$ 

27 // entry point : invocation of TGA for each of the  $n$  network adaptors
28 for  $i = 0 \dots n$  : invoke  $N_{adaptor}.\text{entry\_point}[i]$ 

```

Figure 8. Pseudo Code Description of Gateway Execution Engine

The pseudo code for such an execution is depicted in Figure 9. Lines 1 to 7 contain the declarations of the data structures, which control the TGA execution. A TGA contains clocks  $X$ , variables  $V$ , transitions  $T$ , and the current location  $l$ . A transition goes from location  $l_1$  to location  $l_2$  with a guard, an assignment action, and a communication action. The guard is a boolean condition on clocks, variables, the gateway repository, and ports. The assignment action is a function that updates clocks and variables.

The communication action in Line 6 provides convertible element/variable mappings to transfer the contents of variables into convertible elements (in the gateway repository) and vice versa. A convertible element/variable mapping consists of a reference to a variable of the TGA and a reference to a convertible element. In addition, the communication action provides message/variable mappings (Line 7), each referencing a variable of the TGA and a message. The message/variable mappings are required for the interactions at ports.

```

1   $A = \langle X[], V[], T[], l \rangle$ 
2   $T = \langle l_1, l_2, \varphi, \alpha, \mathcal{G} \rangle$ 
3   $\varphi = (X[], V[], R.c_{state}, R.c_{event}, P.m_{state}, P.m_{event}) \rightarrow (\mathbb{T}, \mathbb{F})$ 
4   $\alpha = (X[], V[]) \rightarrow (X[], V[])$ 
5   $\mathcal{G} = \langle map_{conv}[], map_{msg}[], R.c_{state}.meta, R.c_{event}.meta \rangle$ 
6   $map_{conv} = \langle v^{ref}, c^{ref}, dir \rangle$ 
7   $map_{msg} = \langle v^{ref}, m^{ref}, dir \rangle$ 
8  execution_step:
9  tick = 0
10 while (ticks < P)
11   while ( $\exists$  transition  $T \in A$  with  $T.\varphi = \mathbb{T} \wedge Al = T.l_1$ )
12      $(X[], V[]) = \alpha(X[], V[])$ 
13      $\forall map_{conv} \in T.\mathcal{G}$  with  $map_{conv}.dir = out$  : push( $map_{conv}.c^{ref}$ )
14      $\forall map_{conv} \in T.\mathcal{G}$  with  $map_{conv}.dir = in$  : pull( $map_{conv}.c^{ref}$ )
15      $\forall map_{msg} \in T.\mathcal{G}$  with  $map_{msg}.dir = out$  : send( $map_{msg}.m^{ref}$ )
16      $\forall map_{msg} \in T.\mathcal{G}$  with  $map_{msg}.dir = in$  : receive( $map_{msg}.m^{ref}$ )
17      $\forall c_{state} \in R.c_{state} : c_{state}.meta.b_{req} = \mathbb{T}$ 
18      $\forall c_{event} \in R.c_{event} : c_{event}.meta.b_{req} = \mathbb{T}$ 
19      $Al = T.l_2$ 
20   end
21    $\forall X \in A : X = X + 1$ 
22   ticks = ticks + 1
23 end

```

Figure 9. Pseudo Code Description of Network Adaptor with Execution Step of Timed Gateway Automaton (TGA)

The code for an execution step of the TGA starts in Line 8. Upon each invocation, the TGA is executed for  $P$  ticks, where  $P$  is the number of ticks of the activation period of the gateway middleware. In Line 11, the guards of the transitions at the current location are evaluated. In case of a firing guard, which is uniquely defined since the TGA is deterministic, the execution of assignment and communication actions takes place.

The assignment action updates the values of variables and clocks (Line 12). The communication action beginning at Line 13 invokes the operations push, pull, send and receive provided by the gateway execution engine using the mappings from Lines 6 and 7. Finally, the update request indication associated with the communication action are set in Lines 17 and 18.

After the completion of the communication action, the current location is updated. In the successive location, the process of evaluating guards is repeated, again taking a transition if the corresponding guard fires. This process continues, until no guard fires, but the invariant of the current location holds. In this case, time progresses again by one tick. This process is repeated for  $P$  ticks.

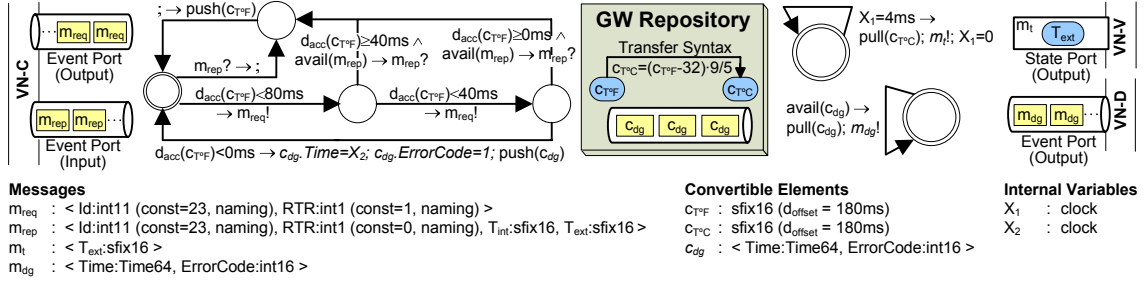


Figure 10. Exemplary Gateway Between an Event-Triggered and a Time-Triggered Virtual Network

## Automatic Generation of Gateway Middleware

The gateway middleware deployed in the DECOS cluster offers a generic execution environment for realizing gateway functionality. The gateway middleware is parameterized by code for the network adaptors and configuration data structures for the gateway execution engine.

Both the code for the network adaptors and configuration data structures are automatically generated from the gateway specification model using a *code generation tool*. The code generation tool is based on the XML C parser toolkit developed for the Gnome project. It takes as input an XML Metadata Interchange (XMI) representation of the gateway specification UML model (see Section 5). The output of the code generation tool are C source files with code for the network adaptors and configuration data structures.

## 7. Example

This section describes an example, which demonstrates the application of the gateway framework and the accompanying development process in a typical automotive scenario.

### Exemplary Distributed Application Subsystems

The following three DASs of a car have been used in conjunction with the exemplary gateway on the prototype DECOS cluster.

**Comfort DAS.** The functionality of this DAS includes the control of the doors, the seats, the climate control, and the lighting of the passenger compartment. A job of the comfort DAS called the measurement job determines the interior and exterior temperature as an input to other jobs of the DAS, such as a climate control job and a job controlling the instrument panel. The temperature measurement job interacts with the other jobs of the DAS using a request/reply communication on the virtual network (called VN-C) of the comfort DAS. Such a request/reply communication is also supported by the CAN protocol (cf. remote data request [Bos91, p. 9]), which is the most-widely used protocol in the automotive industry. Upon the reception of a request message, the temperature measurement job broadcasts a response message containing two real-time images (i.e., interior and exterior values) each denoting the measured temperature in degrees Fahrenheit as a fixed-point number.

In the prototype cluster, the transmission latency of a single message (i.e., either request or reply message) on VN-C is always below 20 ms. This bound has been determined by measuring the end-to-end transmission latency between any two jobs. The bound will be necessary to provide the specification of the network adaptors and the gateway repository (cf. Section 10) with bounded

end-to-end latencies across the gateway and a guaranteed temporal accuracy of the temperature measurements.

**Vehicle-Dynamics DAS.** In the vehicle dynamics DAS all sensory information that is relevant for controlling the dynamics of the vehicle is captured (e.g., yaw rate, roll rate, longitudinal acceleration). The jobs of the vehicle dynamics DAS perform periodic measurements and broadcast the captured real-time images on the time-triggered virtual network (VN-V) of the DAS. Since the sensors in the vehicle-dynamics DAS exhibit temperature-dependent characteristics, the temperature measurements from the comfort DAS can be used to improve the accuracy of the sensors in the vehicle-dynamics DAS. However, for relaying the temperature measurements from VN-C to VN-V, the gateway not only needs to resolve a protocol mismatch (i.e., event-triggered request/reply vs. time-triggered communication), but also convert between different syntactic representations. The jobs in the vehicle-dynamics DAS expect temperature values in degrees Celsius, while degrees Fahrenheit are the unit of measurement in the comfort DAS. Like in VN-C, the transmission latency of a single message between any two jobs on VN-V of the prototype cluster is also bounded by 20 ms.

**Diagnostic DAS.** In present day cars, the detection of a failure through the on-board diagnosis system causes the writing of a breakdown log entry. However, this information is often insufficient to understand the complex processes that caused the subsystem to fail, since typically only local information is provided that does not allow to correlate experienced failures at different parts of the system. For this reason, on-line diagnostic mechanisms within a dedicated diagnostic DAS are useful in order to determine the nature of an experienced fault with respect to a maintenance-oriented fault model [PO06]. An important input for such a diagnostic DAS in the exemplary automotive system is information concerning jobs that violate their specifications. For example, if the temperature measurement job in the comfort DAS fails to send a reply message within the specified time interval after a request message, an error-indication message needs to be sent on the event-triggered virtual network of the diagnostic DAS (i.e., VN-D).

## Specification of the Gateway

The purpose of the gateway is the redirection of messages with temperature measurements and diagnostic information between the virtual networks. The gateway has to perform event-triggered receptions of messages from the temperature measurement job of the comfort DAS using a request/reply protocol. After the extraction of the exterior temperature and the conversion between the different measurement units, the gateway transmits the exterior temperature on the time-triggered virtual network of the vehicle-dynamics DAS. Furthermore, the gateway sends messages with error indications to the diagnostic DAS.

In the following, we will describe the constituting parts of the gateway specification for the automotive example. Figure 10 gives an overview of this gateway and depicts the network binding with the ports towards the accessed virtual networks, the message syntax, the gateway repository, and the specification of the network adaptors comprising three TGA.

**Network Binding.** The network binding names the three virtual networks that are accessed by the gateway. In addition, the ports are specified (e.g., queue lengths for the event-triggered virtual networks) and the messages with their data direction (send or receive) are defined.

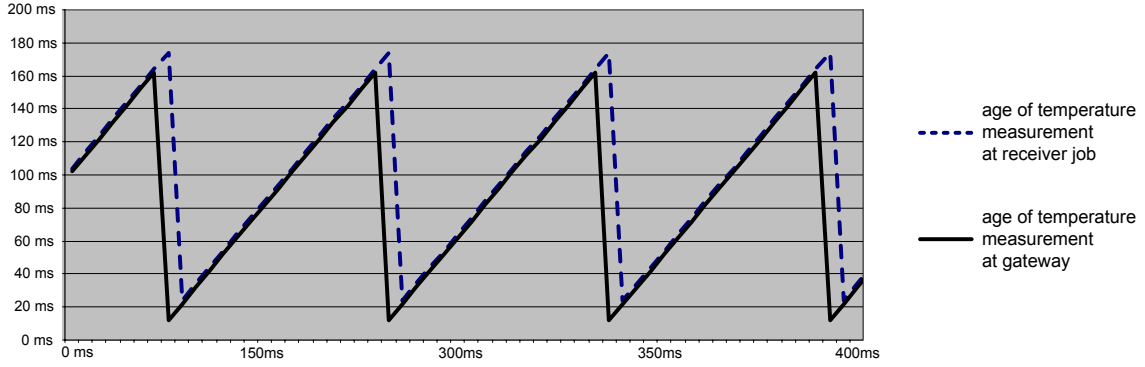


Figure 11. Age of real-time image in gateway repository and at point of use

**Syntactic Specification.** The syntactic part of the gateway specification defines the structure of the received and sent messages and specifies which parts of a message form the message name. The messages received from the virtual network of the comfort DAS conform to the syntax of a standard CAN message [Bos91]. The message name includes the 11-bit CAN identifier, which discriminates the temperature measurements from other received messages. The user data consist of two 16-bit signed fixed point numbers denoting the interior and exterior temperature. A message, which is sent on the virtual network of the vehicle-dynamics DAS contains a 16-bit signed fixed point number. This messages uses an implicit name, i.e., the port uniquely identifies the messages. A message, which is sent on the virtual network of the diagnostic DAS, provides information on the type of the detected error (i.e., aggregate *ErrorCode*) and a timestamp w.r.t. to the global time base.

**Definition of the Gateway Repository.** The gateway repository contains three convertible elements. Two convertible elements with state semantics store exterior temperature measurements, one with values in degrees Fahrenheit ( $c_{T^{\circ}F}$ ) as acquired from VN-C and the other one with values in degrees Celsius ( $c_{T^{\circ}C}$ ) destined to VN-V. In the example we assume that the dynamics of the environment cause a temperature measurement to remain accurate for 200 ms. Since it takes a maximum of 20 ms in the prototype cluster for a measurement to arrive via VN-V at the consumer, each of these two convertible elements has a temporal accuracy offset of 180 ms. The third convertible element contains error indications for VN-D. It exhibits event semantics and is represented as a queue in the gateway repository.

**Specification of Network Adaptors.** The specification of the network adaptors consists of three TGA. The TGA in the left of Figure 10 serves for the interaction with VN-C. This TGA has to keep the convertible element  $c_{T^{\circ}F}$  with the temperature value temporally accurate in the gateway repository. For this purpose, the TGA sends a request message  $m_{\text{req}}$  on VN-C in case the temporal accuracy interval of the temperature value in the gateway repository is lower than 80 ms. The maximum latency of a request message or a response message is 20 ms. Therefore, if a reply message  $m_{\text{rep}}$  fails to arrive within 40 ms, another request message is transmitted. If the reply fails to arrive within 40 ms after the second request message, the TGA pushes a convertible element called  $c_{\text{dg}}$  with an error indication into the gateway repository. In this case, the temporal accuracy of the temperature value in the gateway repository is lost due to a failure of the measurement

job failure. In case a reply message arrives within one of the two 40 ms timeout intervals, the convertible element  $c_{T^{\circ}F}$  with the exterior temperature is pushed into the gateway repository.

The second TGA, which is located on the right-top side in Figure 10, periodically transmits the temperature value from the gateway repository on VN-V at a priori specified global points in time. Prior to a transmission, the TGA pulls the convertible element with the temperature value  $c_{T^{\circ}C}$  from the gateway repository.

The third automaton on the right-bottom side in Figure 10, relays the error-indications concerning the temperature measurement job to the diagnostic DAS. For this purpose, the TGA determines the availability of convertible elements (i.e.,  $\text{avail}(c_{dg})$ ). If a convertible element is available, it is sent on VN-D.

**Transfer Syntax.** The transfer syntax is an internal rule for converting from one syntactic representation to another one. Due to an assumed syntactic property mismatch in the automotive example (i.e., different units of measurement used in the DASs), the transfer syntax converts from degrees Fahrenheit to degrees Celsius.

## Instantiated Gateway Services on the Prototype Cluster

The exemplary gateway specification has been expressed in compliance to the meta-model in Section 5 and used as an input for the code generation tool. In the following we will describe the performance and resource consumption of the automatically generated gateway code.

**Temporal Accuracy.** The main purpose of the exemplary gateway is to ensure that the jobs in the vehicle-dynamics DAS receive temporally accurate temperature measurements. By sending request messages for temperature measurements, the TGA for the interaction with VN-C prevents the convertible element from losing temporal accuracy.

The age of the temperature measurements over time is depicted in Figure 11. At the gateway the age of the measured value varies between 12 ms and 162 ms. The lower bound (12 ms) is the transmission latency from the measurement job to the gateway. After the gateway has requested an update of the temperature value, the measurement job samples a temperature at a sampling point and transmits a response message. The delay between the sampling point and the arrival of the response message at the gateway is 12 ms. The delay of 12 ms is no consequence of the TGA, but results from the timing of the event-triggered VN and the scheduling of the measurement job and the gateway middleware.

The upper bound (162 ms) is determined by the TGA, namely by the decision on when to issue an update request. The TGA issues an update request when the temporal accuracy drops below 80 ms, i.e., the age is larger than 100 ms. The measurement value is updated following a delay for the arrival of the request message at the measurement job and the 12 ms for the arrival of the response message. Hence, the observed delays conform to the latency bound (i.e., 20 ms on VN-C and VN-V) used in the specification of the gateway.

In order to understand the delays that have been observed in the implementation of the DECOS cluster, we will also describe the delays analytically based on the communication schedule and the computational schedule (cf. Figure 12). The communication schedule defines the layout of the TDMA slots for the different nodes on the time-triggered physical network. The prototype cluster employs a time-triggered communication schedule, in which every node sends exactly once during each TDMA round. Each node  $i$  sends during slot  $i$ .

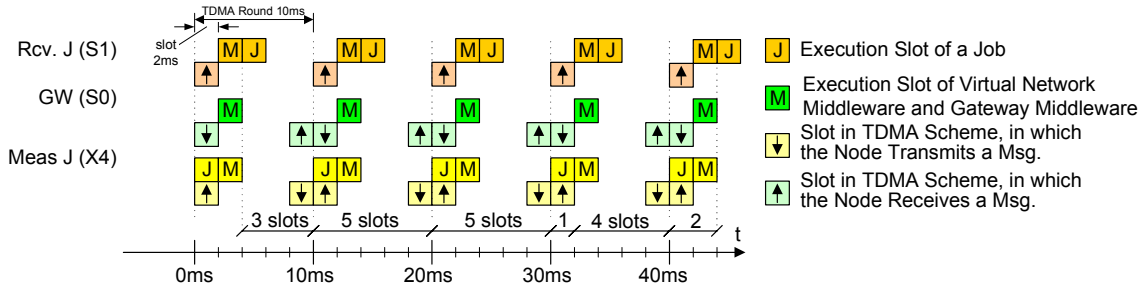


Figure 12. Scheduling of tasks and communication

The computational schedule controls the execution slots for the measurement job in the comfort DAS, the receiver job in the vehicle-dynamics DAS, and the gateway middleware. The temperature measurement job is located in node 4 and scheduled during slot 0. The middleware task implementing the gateway is located in node 0 and scheduled during slot 1. The exemplary consumer job within the vehicle-dynamics DAS is located in node 1 and scheduled during slot 2.

Using this information, we can compute the latencies for the messages that are exchanged as part of an information transfer across the gateway between the temperature measurement job and the receiver job in the vehicle-dynamics DAS.

- **Transmission latency on VN-C of a request message from the gateway to the temperature measurement job.** It takes 3 slots until the message is sent on the time-triggered physical network. It takes 5 slots from the sending slot of node 0 on the time-triggered physical network until the temperature measurement job is scheduled and can process the request. The end-to-end delay for the request message between the gateway and the temperature measurement job is thus 8 slots (i.e., 16 ms).
- **Transmission latency on VN-C of the response message from the temperature measurement job to the gateway.** It takes 5 slots until the message is sent on the time-triggered physical network. It takes 1 slot from the sending slot of node 4 on the time-triggered physical network until the gateway middleware task is scheduled and can process the response. The end-to-end delay for the request message between the gateway and the temperature measurement job is thus 6 slots (i.e., 12 ms).
- **Transmission latency on VN-V of the message from the gateway to the consumer job.** It takes 4 slots until the message with the temperature value is sent on the time-triggered physical network. It takes 2 slot from the sending slot of node 0 on the time-triggered physical network until the exemplary receiver job is scheduled and can process the response. The end-to-end delay for the request message between the gateway and the temperature measurement job is thus 6 slots (i.e., 12 ms).

**Resource Requirements.** For the generated gateway on the Soekris single board computer net4521, Figure 13 gives an overview of the maximum execution time and memory consumption broken down into the three TGA. The maximum execution times of the TGA are important parameters, because the maximum execution time of the entire gateway middleware task must be smaller than the execution slot provided by the time-triggered scheduler of the execution environment.



Gateway Element	Max. Execution Time	Memory Consumption
TGA for VN-C	25 $\mu$ s	3.1 KB
TGA for VN-V	29 $\mu$ s	2.1 KB
TGA for VN-D	21 $\mu$ s	2.4 KB

Figure 13. Execution Times of TGAs

## 8. Discussion

The use of gateways for the interconnection of networks with different communication protocols is an important problem that has received much attention in previous work. Many authors have focused on formal specifications based on communicating finite state machines. This paper describes a novel solution for the realization of gateways based on a real-time database in-between the interconnected networks. The real-time database stores temporally accurate real-time images in conjunction with meta information (e.g., instant of most recent update, information w.r.t. to update requests). The major benefit of the real-time database is the ability for a constructive realization of gateways in distributed real-time systems. Large, complex gateways can be divided into smaller modules, which are not only simpler but facilitate reuse and localize changes. For each network, developers can independently specify which messages update the real-time database and which messages are sent with the information from the real-time database. The introduced timed gateway automata provide a powerful and intuitive formalism for this task. They enable developers to specify the protocols for accessing specific networks along with the corresponding syntax and naming transformations.

The presented prototype implementation demonstrates that the presented gateways can be effectively used in an exemplary automotive scenario. Tools take a formal specification constrained by a UML meta-model as an input and perform automatic code generation of the gateway software for the prototype cluster.

## Acknowledgments

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

## References

- [AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.
- [Aer06] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Specification 653: Avionics Application Software Standard Interface, Part 1 - Required Services*, March 2006.
- [Aue89] J. Auerbach. A protocol conversion software toolkit. In *SIGCOMM '89: Symposium Proc. on Communications architectures & protocols*, pages 259–270, New York, NY, USA, 1989. ACM Press.
- [Bir03] S. Birch. Pre-safe headlines S-Class revisions. *Automotive Engineering International*, pages 15–18, January 2003.
- [Bos91] Robert Bosch GmbH, Stuttgart, Germany. *CAN Specification, Version 2.0*, 1991.
- [CFMB98] Y. Chawathe, S.A. Fink, S. McCanne, and E.A. Brewer. A proxy architecture for reliable multicast in heterogeneous environments. *ACM Multimedia*, pages 151–159, 1998.
- [CG02] C.R. Carlson and J.C. Gerdes. Practical position and yaw rate estimation with GPS and differential wheel-speeds. In *Proc. of 6th Int. Symposium on Advanced Vehicle Control*, 2002.
- [Dod01] D.S. Dodge. Gateways - 101. In *Proc. of the Military Communications Conference*, volume 1, pages 532–538. IEEE, October 2001.

- [Fle05] FlexRay Consortium. *FlexRay Communications System Protocol Specification Version 2.1*, May 2005.
- [FPT05] R.C. Ferguson, B.L. Peterson, and H.C. Thompson. System software framework for system of systems avionics. In *Proc. of the 24th Digital Avionics Systems Conference*, volume 2, pages 8.A.1–1 – 8.A.1–10, October 2005.
- [HPOS05] B. Huber, P. Peti, R. Obermaisser, and C. El Salloum. Using RTAI/LXRT for partitioning in a prototype implementation of the DECOS architecture. In *Proc. of the Third Int. Workshop on Intelligent Solutions in Embedded Systems*, May 2005.
- [IAC99] S. Iren, P.D. Amer, and P.T. Conrad. The transport layer: tutorial and survey. *ACM Computing Surveys (CSUR)*, 31(4):360–404, 1999.
- [KG94] H. Kopetz and G. Grunsteidl. TTP – A protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, 1994.
- [KK90] H. Kopetz and K. H. Kim. Temporal uncertainties in interactions among real-time objects. In *Proc. of Ninth Symposium on Reliable Distributed Systems*, pages 165–174, Huntsville, AL, USA, October 1990.
- [KLNS91] D.M. Kristol, D. Lee, A.N. Netravali, and K.K. Sabnani. Efficient gateway synthesis from formal specifications. *SIGCOMM Comput. Commun. Rev.*, 21(4):89–97, 1991.
- [Kop97] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.
- [LH02] G. Leen and D. Heffernan. Expanding automotive electronic systems. *Computer*, 35(1):88–93, January 2002.
- [Nex03] NextTTA. Project deliverable D2.4. Emulation of CAN and TCP/IP, September 2003. IST-2001-32111. High Confidence Architecture for Distributed Control Applications.
- [OMG02] OMG. Smart Transducers Interface. Specification ptc/2002-05-01, Object Management Group, May 2002. Available at <http://www.omg.org/>.
- [OP05] R. Obermaisser and P. Peti. Realization of virtual networks in the DECOS integrated architecture. In *Proc. of the Workshop on Parallel and Distributed Real-Time Systems 2006 (WPDRTS)*. IEEE, April 2005.
- [PO06] P. Peti and R. Obermaisser. A diagnostic framework for integrated time-triggered architectures. In *Proc. of the 9th IEEE Int. Symposium on Object-oriented Real-time distributed Computing*, April 2006.
- [RM91] M. Rajagopal and R.E. Miller. Synthesizing a protocol converter from executable protocol traces. *IEEE Transactions on Computers*, 40(4):487–499, April 1991.
- [Rus01] J. Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, SRI Intern., 2001.
- [Sim96] H.A. Simon. *The Sciences of the Artificial*. MIT Press, 1996.
- [SLO03] M. Segarra, T. Losert, and R. Obermaisser. Hard real-time CORBA: TTP transport definition. Technical Report IST37652/067, Universidad Politecnica de Madrid, Lunds Tekniska Högskola, Technische Universität Wien, SCILabs Ingenieros, March 2003.
- [SSN98] J.A. Stankovic, S.H. Son, and C.D. Nguyen. The cogency monitor: an external interface architecture for a distributed object-oriented real-time database system. In *Proc. of the Fourth IEEE Real-Time Technology and Applications Symposium*, pages 71–78, June 1998.
- [WP99] D. Wybo and D. Putti. A qualitative analysis of automatic code generation tools for automotive powertrain applications. In *Proc. of the 1999 IEEE Int. Symposium on Computer Aided Control System Design*, pages 225–230, Hawaii, USA, August 1999.

# A TRANSIENT-RESILIENT SYSTEM-ON-A-CHIP ARCHITECTURE WITH SUPPORT FOR ON-CHIP AND OFF-CHIP TMR

*Proceedings of the 7th European Dependable Computing Conference, pages 123–134, Kaunas, Lithuania, May 2008.*

Roman Obermaisser  
Vienna University of Technology  
Real-Time Systems Group  
romano@vmars.tuwien.ac.at

Hubert Kraut  
Vienna University of Technology  
Real-Time Systems Group  
peti@vmars.tuwien.ac.at

Christian Salloum  
Vienna University of Technology  
Real-Time Systems Group  
peti@vmars.tuwien.ac.at

**Abstract** The ongoing technological advances in the semiconductor industry make MPSoCs more attractive, because uniprocessor solutions do not scale satisfactorily with increasing transistor counts. In conjunction with the increasing rates of transient faults in logic and memory associated with the continuous reduction of feature sizes, this situation creates the need for novel MPSoC architectures. This paper introduces such an architecture, which supports the integration of multiple, heterogeneous IP cores that are interconnected by a time-triggered NoC. Through its inherent fault isolation and determinism, the proposed MPSoC provides the basis for fault tolerance using Triple Modular Redundancy (TMR). On-chip TMR improves the reliability of a MPSoC, e.g., by tolerating a transient fault in one of three replicated IP cores. Off-chip TMR with three MPSoCs can be used in the development of ultra-dependable applications (e.g., X-by-wire), where the reliability requirements exceed the reliability that is achievable using a single MPSoC. The paper quantifies the reliability benefits of the proposed MPSoC architecture by means of reliability modeling. These results demonstrate that the combination of on-chip and off-chip TMR contributes towards building more dependable distributed embedded real-time systems.

## 1. Introduction

Due to diminishing returns from uniprocessor optimizations, chip designers are facing the need for new computer architectures. According to Pollack's rule [Gel01], the increase in performance of a uniprocessor is only about the square root of the increase in the number of devices, which implies that doubling the transistor count will lead to a performance improvement of about 40%.

Multi-core architectures in the form of Multi-Processor System-on-a-Chips (MPSoCs) are a solution to circumvent Pollack's rule. If an application can be partitioned into a set of nearly autonomous concurrent functions, then a nearly linear performance improvement could be achieved by assigning a dedicated processing element to each of these concurrent functions. MPSoCs combine multiple heterogeneous processing elements, which can be interconnected by a NoC. However, the key to high performance in multi-processor systems-on-a-chip are applications with inherent parallelism. Fortunately, the inherent concurrency in a typical embedded application (e.g., automotive electronics, avionics) satisfies this requirement.

Another challenge in the development of new processors is the increasing importance of transient faults. The types and causes of failures for electronics have changed over the years. Failure analysis in recent years has revealed that permanent failures have been reduced by improvements in technology but due to the higher level of complexity and downsizing other failure classes have emerged. The tremendous improvements made by the IC industry with respect to permanent failure rates are extenuated by increasing transient failure rates for instance due to semiconductor process variations, shrinking geometries, and lower power voltages [MPG02, Con02]. These result in higher sensitivity to neutron and alpha particles, and consequently have an impact on dependability by increasing the transient failure rates [GAM<sup>+</sup>02]. These technological effects affect in particular upcoming generations of MPSoCs manufactured using Very Deep Sub-Micron (VDSM) processes [Sem06].

Due to these technological constraints, there is a growing importance of self correcting intelligence embedded into MPSoCs specifically targeting transient faults. Solutions for self correcting intelligence have emerged at different abstraction levels. For example, in [GSVP03] chip-level redundancy through a redundantly threaded multiprocessor with recovery is presented. This solution provides fault tolerance by replicating an application into two communicating threads, namely a leading thread executing ahead of a so-called trailing thread. The execution of the trailing threads is delayed by a predefined number of instructions in order to enable the trailing thread to use memory load values and branch outcomes of the leading thread.

Another approach for tolerating transient faults at chip-level is time redundancy [AN00, DNR02, Nic99]. Detection and masking of transient faults occurs based on the comparison of the computational results that are gained through redundant computations performed in temporal succession. Disadvantages of time redundancy are a performance penalty and constraints on the duration of the transient pulse that can be detected and corrected.

The focus of this paper are distributed real-time systems, where the correct behavior depends not only on the logical results but also on the points in time at which these results are produced. This real-time requirement particularly applies to fault scenarios. Hence, the computer system has to be fail operational by continuing to provide the specified application services (e.g., engine control in a car) in the value and time domain despite the occurrence of transient faults.

Consequently, we have selected spatial redundancy using Triple Modular Redundancy (TMR) for the proposed transient-resilient MPSoC architecture. The temporal behavior of a TMR configuration does not change as long as only one of the replicas is affected by a transient fault.

In contrast to other solutions employing TMR within a chip (e.g., LEON3FT [Gai06]), we perform replication of complete IP cores instead of introducing spatial redundancy at transistor level or at the level of logic circuits. The decision of selecting TMR at the level of IP cores is motivated by the following reasons:

- *Superior time and energy efficiency.* The voting in a TMR configuration involves an inevitable overhead. For TMR at the level of small logic circuits, this overhead occurs when-

ever this logic circuit is used. For example, in [AAN00] an overhead of up to 23% has been determined for voting in TMR in different types of arithmetic circuits. Therefore, the solution presented in this paper performs incoming voting at a coarser level. Voting occurs on the final computational results of IP cores and not on the intermediate computational results. For example, consider an IP core for FFT that is implemented in a TMR configuration. Only the final Fourier coefficients are voted upon and not the results of each intermediate computation. Hence, there is less timing overhead and energy consumption.

- *Ability to use standard libraries.* Voting at the level of IP cores allows to use standard IP core libraries. In contrast, the introduction of fault-tolerant logic circuits (e.g., fault tolerant adder [AAN00]) would require developers to construct completely new IP cores.
- *Higher resilience against spatial proximity faults.* TMR at the level of IP cores enables designers to physically separate the replicas on the chip. Hence, the probability of correlated transient faults affecting multiple replicas is significantly diminished. According to [Con03, SKK<sup>+</sup>02] the probability of particle-induced multi-bit errors and soft errors within the combinatorial logic is going to increase.
- *Foundation for design diversity and heterogeneity.* The proposed solution allows to exploit diverse replicas, i.e., different IP cores that provide the same services, while having been designed by different developers in order to reduce the probability of design faults. For example, a TMR configuration with three different implementations of a control algorithm would allow to tolerate a design fault restricted to one of the three replicas.

In this paper, we present a System-on-a-Chip (SoC) architecture that supports both on-chip and off-chip TMR. For on-chip TMR, the replicas are three IP cores that are interconnected by a NoC. Incoming voting occurs at the IP cores that receive the redundant messages from the replicas.

The major contributions of the paper are as follows:

- *SoC architecture with inherent support for transient-resilience through TMR.* The presented architecture provides the foundations for TMR (e.g., by being design for replica determinism [Pol96]). Furthermore, generic IP cores for voting enable designers to rapidly construct TMR configurations.
- *Complementation of on-chip and off-chip TMR.* Redundancy only leads to reliable systems if the constituting replicas are reliable [LV62]. Hence, the effectiveness of off-chip TMR can be improved when increasing the reliability of individual chips using on-chip TMR. This complementarity of on-chip and off-chip TMR is quantitatively evaluated in the paper through reliability modeling.
- *Spatial redundancy at level of IP cores.* Raising the granularity for TMR from logic circuit to IP core level brings the above mentioned benefits, such as superior time and energy efficiency, the ability to use standard libraries, higher resilience against spatial proximity faults, and support for design diversity.

The paper is structured as follows. Section 2 gives an overview of the proposed SoC architecture. The realization of on-chip and off-chip TMR is the focus of Section 3. Section 4 describes a reliability model of the different TMR configuration. Using the Moebius tool, we quantitatively analyze the reliability of an individual SoC and of clusters with multiple SoCs. The results of this analysis are presented in Section 5.

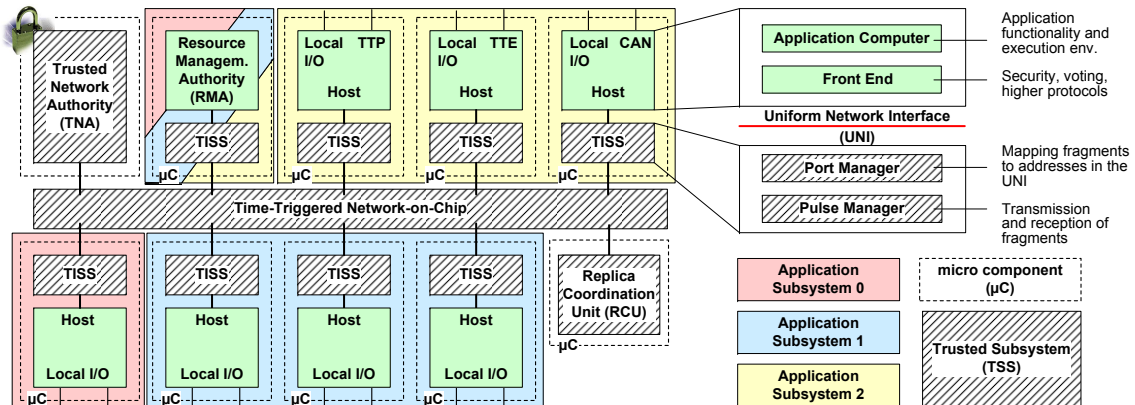


Figure 1. Structure of Time-Triggered SoC Architecture

## 2. Time-Triggered System-on-a-Chip Architecture

The central element of the presented SoC architecture is a time-triggered NoC that interconnects multiple, possibly heterogeneous IP blocks called micro components (see Figure 1). The SoC introduces a *trusted subsystem*, which ensures that a fault (e.g., a software fault) within the host of a micro component cannot lead to a violation of the micro component's temporal interface specification in a way that the communication between other micro components would be disrupted. For this reason, the trusted subsystem prevents a faulty host within a micro component from sending messages during the sending slots of any other micro component.

Furthermore, the time-triggered SoC architecture supports integrated resource management. For this purpose, dedicated architectural elements called the Trusted Network Authority (TNA) and the Resource Management Authority (RMA) accept resource allocation requests from the micro components and reconfigure the SoC, e.g., by dynamically updating the time-triggered communication schedule of the NoC.

### Micro Component

The introduced SoC can host multiple application subsystems (possibly of different criticality levels), each of which provides a part of the service of the overall system. An example of an application subsystem in the automotive domain would be a braking subsystem. A nearly autonomous and possibly heterogeneous Intellectual Property (IP)-block, which is used by a particular application subsystem is denoted as a *micro component*. A micro component is a self-contained computing element, e.g., implemented as a general purpose processor or as special purpose hardware. An application subsystem can be realized on a single micro component or by using a group of possibly heterogeneous micro components (either on one or multiple interconnected SoCs).

The interaction between the micro components of an application subsystem occurs solely through the exchange of messages on the time-triggered NoC. Each micro component is encapsulated, i.e., the behavior of a micro component can neither disrupt the computations nor the communication performed by other micro components. Encapsulation prevents by design temporal interference (e.g., delaying messages or computations in another micro component) and spatial interference (e.g., overwriting a message produced by another micro component). The only man-

ner, in which a faulty micro component can affect other micro components, is by providing faulty input to other micro components of the application subsystem via the sent messages.

Due to encapsulation, the SoC architecture supports the detection and masking of such a failure of a micro component using TMR. Encapsulation is necessary for ensuring the independence of the replicas. Otherwise, a faulty micro component could disrupt communication or communication of the replicas, thus causing common mode failures.

Also, encapsulation is of particular importance for the implementation of SoCs encompassing application subsystems of different criticality levels. In such a mixed criticality system, a failure of micro components of a non safety-critical application subsystem must not cause the failure of application subsystems of higher criticality.

For the purpose of encapsulation, a micro component comprises two parts: a *host* and a so-called *Trusted Interface Subsystem (TISS)*. The host implements the application services. Using the TISS, the time-triggered SoC architecture provides a dedicated architectural element that protects the access to the time-triggered NoC.

## Requirements for the Time-Triggered Network-on-a-Chip

The time-triggered NoC interconnects the micro components of an SoC. The purposes of the time-triggered NoC encompass clock synchronization for the establishment of a global time base, as well as the predictable transport of periodic and sporadic messages.

**Clock Synchronization.** The time-triggered NoC performs clock synchronization in order to provide a global time base for all micro components despite the existence of multiple clock domains. The time-triggered NoC is based on a uniform time format for all configurations, which has been standardized by the OMG in the smart transducer interface standard [OMG02].

**Predictable Transport of Messages.** Using TDMA, the available bandwidth of the NoC is divided into periodic conflict-free sending slots. We distinguish between two utilizations of a periodic time-triggered sending slot by a micro component. A sending slot can be used for the *periodic transmission of messages* or the *sporadic transmission of messages*. In the latter case, a message is only sent if the sender must transmit a new event to the receiver. If no event occurs at the sender, the reserved bandwidth remains available.

The allocation of sending slots to micro components occurs using a communication primitive called *pulsed data stream* [H.K06]. A pulsed data stream is a time-triggered periodic unidirectional data stream that transports data in pulses with a defined length from *one* sender to *n* a priori identified receivers at a specified phase of every cycle of a periodic control system.

A pulse consists out of one or more *fragments* of variable size, which are sent successively during the duration of a single pulse. The defining parameters of a pulsed data stream (i.e., pulse period, pulse phase) determine the allocation of TDMA slots to the micro component that sends the pulsed data stream. TDMA slots are allocated to the sender micro component in a time interval, which periodically recurs with the pulse period and has a length controlled by the pulse duration.

## Host

The host performs the computations that are required to deliver the intended application service of a micro component. In general, the host is not assumed to be free of design faults. The host is divided into an *application computer*, which implements the intended application services and the so-called *front end*. The front-end provides a domain-specific network interface to the ap-

plication computer (e.g., temporal firewall interface [KN97] or event queues). It can incorporate middleware bricks that extend the communication services that are provided by the TISS. Examples of such services are security functionality (e.g., encryption, authentication) or fault-tolerance mechanisms (e.g., voting).

### Trusted Interface Subsystem

The TISS manages the host's access to the underlying 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 micro component. Since the table cannot be modified by the host, a design fault or a hardware fault restricted to the host of a micro component cannot affect the exchange of messages by other micro components.

As depicted in Figure 1, the TISS is structured into two parts, namely a port manager and a pulse manager. The pulse manager is responsible for transmitting fragments of pulses, while providing media access control and logical link control. The port manager maps each fragment that is received or sent by the pulse manager to a corresponding address in the uniform network interface.

**Port Manager.** The port manager implements the transport layer in the OSI reference model [Int94]. The port manager maps each fragment that is received or sent by the pulse manager to a corresponding address in the uniform network interface. Furthermore, the port manager manages the control fields of the ports (e.g., write and read position of event ports, synchronization fields for state ports). In addition to the ports, the port manager provides a programmable timer-interrupt service, a watchdog service and a power control service.

**Pulse Manager.** The pulse manager accesses the TTNoC by sending and receiving single fragments of a pulsed data stream according to the message definitions stored in the MEDL. Whenever a fragment is received by the pulse manager the fragment number which identifies the fragment within a message, the associated port number and the fragment itself are handed over to the port manager. On the other hand, when a fragment is scheduled to be transmitted by the pulse manager, the pulse manager issues a request to the port manager which includes the fragment number and the port number of the requested fragment. The port manager will then pass the requested fragment to the pulse manager.

### Architectural Elements for Resource Management

The purpose of the integrated resource management in the SoC architecture is to dynamically assign computational resources (i.e., micro components) to application subsystems and to grant communication resources and power to the individual micro components.

We distinguish two fundamentally different types of application subsystems: *Safety-critical applications subsystem* need to be certified to the highest criticality classes (e.g., class A according to DO-178B). Non safety-critical applications subsystems, on the other hand, do not require certification to the highest criticality classes. In general, these two types of applications subsystems will involve fundamentally different design paradigms. The focus of safety-critical applications lies on simplicity and determinism in order to facilitate thorough verification and validation. In contrast, non safety-critical applications can provide more complex application services (e.g., need to deal with insufficient a priori knowledge about the environment) and dynamism to handle the challenges of evolving application scenarios and changing environments.



The architectural elements for resource management follow this bivalent distinction of application subsystems. We provide two different architectural elements for enabling integrated resource management, namely the Trusted Network Authority (TNA) and the Resource Management Authority (RMA). The RMA computes new resource allocations for the non safety-critical application subsystems, while the TNA ensures that the new resource allocations have no adverse effect on the behavior of the safety-critical application subsystems. As depicted in Figure 1 the TNA is part of the trusted subsystem of the SoC, whereas the RMA is not. By splitting the entire resource management into two separate parts, where only one is part of the trusted subsystem, the certification of the time-triggered SoC is significantly simplified, since the checking of the correctness of a resource allocation through the TNA is significantly simpler than its generation at the RMA.

### Replica Coordination Unit

The purpose of the Replica Coordination Unit (RCU) is the detection of host failures caused by transient faults and the subsequent resetting of the faulty hosts. The resetting of hosts through the RCU is part of the architectural mechanisms for recovering from transient faults. The RCU detects host failures by comparing the redundant computational results in TMR configurations. In case a divergence of these results occurs, the RCU executes a reset operation at the TISS of the respective micro component.

### Gateways

The proposed SoC architecture supports gateways for accessing chip-external networks (e.g., TTP [KG94] or TTE [KAGS05] in Figure 1). The benefits of gateways include the ability to interconnect multiple SoCs to a distributed system, which enables applications based on the SoC architecture for ultra-dependable systems [SWH95]. Since component failure rates are usually in the order of  $10^{-5}$  to  $10^{-6}$  (e.g., [PMH98] uses a large statistical basis and reports 100 to 500 failures out of 1 Million ECUs in 10 years), ultra-dependable applications require the system as a whole to be more reliable than any one of its components. This can only be achieved by utilizing fault-tolerant strategies that enable the continued operation of the system in the presence of component failures.

In case the chip-external network is also time-triggered (e.g., TTP [KG94], TTE [KAGS05]), the TDMA scheme of the NoC can be synchronized with the TDMA scheme of the chip-external network. The periods and phases of the relayed pulsed data streams on the NoC can be aligned with the transmission start instants of the messages on the time-triggered chip-external network. Consequently, a message that is sent on the chip-external network is delivered to the micro components within a bounded delay with minimum jitter (only depending on the granularity of the global time base). The alignment between pulsed data streams and messages on time-triggered networks ensures that replicated SoCs perceive a message at the same time, i.e., within the same inactivity interval of the global sparse time base [Kop92]. This property is significant for achieving replica determinism [Pol94] as required for active redundancy based on exact voting. Without synchronization between the NoC and the chip-external network, there could always occur a scenario in which one SoC forwards the message to the micro components in one period of the pulsed data stream, while another SoC would forward the message in the next period.

### 3. Fault Tolerance

The Time-Triggered System-on-a-Chip (TTSoC) architecture employs micro components in TMR configurations in order to tolerate transient faults. We denote three replicas in a TMR configuration a Fault-Tolerant Unit (FTU). In addition, we consider the voter at the input of a micro component and the micro component itself as a self-contained unit, which receives the replicated inputs and performs voting by itself without relying on an external voter. We call this behavior *incoming voting*.

With respect to voting, one can differentiate between two kinds of strategies: exact voting and inexact voting [Kop97]. Exact-voting is preferred in the TTSoC architecture, because it is transparent to applications and generic (i.e., independent from any specific application service). The underlying assumption of exact voting is, that the replicas show *replica-deterministic* behavior [Pol96]. Replica determinism requires that all correct replicas produce exactly the same output messages that are at most an interval of  $d$  time units apart (as seen by an omniscient outside observer). In a time-triggered system, the replicas are considered to be replica deterministic, if they produce the same output messages at the same global ticks [Kop97]. A common source of replica indeterminism is in an inconsistent message reception order at the replicas within an FTU.

A TMR-based FTU is a fail-operational component, which means that it continues to deliver a correct service despite the failure of a single replica. Nevertheless, the reliability of an FTU is reduced, if a failed replica remains in an erroneous state. In fact, the reliability of a TMR-based FTU in which one of the three replicas has already failed is actually lower than the reliability of a single replica since the FTU requires both remaining replicas to stay operational. In order to obtain the original reliability of an FTU after the failure of a replica, one of the following actions can be taken:

- In case of a transient fault where the hardware is still operational, a valid state in the replica can be reestablished by adequate recovery mechanisms.
- In case of a permanent hardware fault, the hardware of the replica can be repaired or replaced during the next scheduled maintenance service.
- In case of a permanent hardware fault where a repair action cannot be taken due to temporal or physical constraints (e.g., a faulty IP core within a chip cannot be repaired), the role of the failed replica can be taken over by a spare IP core.

In the following, we will explain TMR-based FTUs at the on-chip and off-chip level. This description will focus on the following categories:

- *Encapsulation*: Common mode failures have to be avoided by *encapsulation* mechanisms that establish a dedicated FCR for each replica. *An FCR is a collection of components that operates correctly regardless of any arbitrary logical or electrical fault outside the region [JR94]*. Without encapsulation it cannot be guaranteed that the replicas fail independently. If a shared communication medium is employed, *error containment* mechanisms (e.g., the bus guardian in TTP [KG94]) are needed to ensure that a fault in a replica cannot disrupt the communication of the other replicas.
- *Replica determinism* has to be supported by the architecture to ensure that the replicas of an FTU observe the received messages in a consistent order.

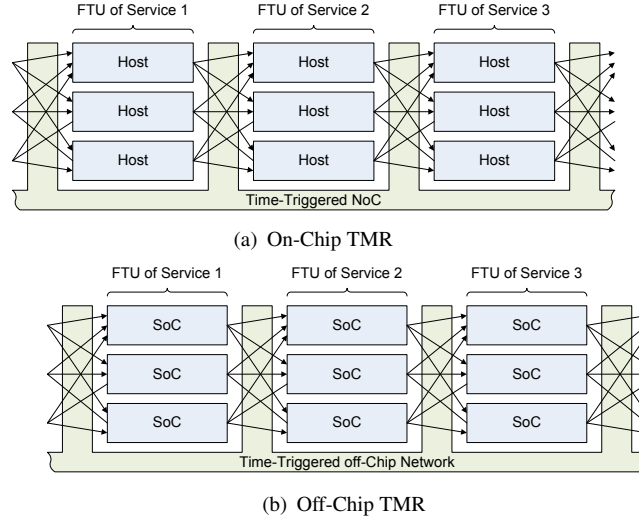


Figure 2. Fault Tolerant Units

- *Temporal predictability*: The communication service has to be *temporally predictable* in order to guarantee that the resulting system can meet its deadlines even in peak-load scenarios.
- *Recovery and repair*: The architecture should provide adequate *recovery* and *repair* mechanisms, in order to reestablish the original reliability of an FTU after the failure of a replica.

## On-chip TMR

The purpose of on-chip TMR is to increase the reliability of services residing on a single chip. For ultra-high reliability, however, off-chip TMR has to be employed (cf. Section 3). Therefore, FTUs are constructed by mapping the replicas to individual hosts which are interconnected by the time-triggered NoC (see Figure 2(a)). The following paragraphs describe how the required architectural properties for TMR are met w.r.t. on-chip TMR.

**Encapsulation.** The FCRs for *physical faults* are the individual hosts and the Trusted Subsystem (TSS) which consists of the NoC, the TISSs and the TNA. Contrary to an operating system that provides dedicated FCRs for multiple tasks on a single processor by complex memory protection mechanisms and preemptive scheduling strategies, the TTSoC architecture naturally provides fault containment by the physical separation of the individual hosts and the TSS.

The independence of the host-FCRs is guaranteed by the fact that hosts can interact with each other exclusively via the exchange of messages over the NoC. There are no other hidden channels (e.g., shared memory) through which a host can interfere with any other host.

The independence of the TSS-FCR is guaranteed by the fact that the hosts have no possibility to directly interfere with the operation of the TSS. The TSS transports messages from one host to another host according to a predefined time-triggered message schedule. This schedule can be exclusively configured by the TNA which is itself part of the TSS. Thus, it is guaranteed that a faulty host cannot disrupt the communication among other hosts.

Contrary to fault-tolerant off-chip networks like TTP [KG94] which are partitioned into multiple FCRs, the TSS is a single atomic FCR, which means that a failure of one of its elements (i.e., the NoC, the TNA or one of the TISSs) can potentially cause a failure of the entire chip. Since in a typical SoC, the die area consumed by the TSS is relatively small compared to the area consumed by the rest of the chip, we expect the failure rate of the TSS to be relatively low (cf. Section 5). Therefore, for on-chip TMR we assume that the TSS does not fail during the mission time of the system.

To conclude, the underlying assumptions for on-chip TMR with respect to physical faults, are that the hosts fail independently and that the TSS does not fail during the system's mission time. Considering the fact, that a single chip is susceptible to common mode failures caused by disruption of the single power supply, particle induced multi-bit errors, extensive EMI disturbances or physical damage, the assumption coverage for these assumptions will not satisfy the requirements for ultra-high dependable systems. Nevertheless, for many applications with less stringent dependability requirements, on-chip TMR can be a cost-effective alternative to increase the reliability of systems realized on a single SoC. Furthermore, in safety-critical systems on-chip TMR can be used in conjunction with off-chip TMR to further improve reliability.

With respect to *design faults*, a host constitutes an FCR as long as no piece of the design is used in any other host. If pieces of a design (e.g., library functions or IP cores) are used in a set of hosts, the entire set has to be considered as a single atomic FCR. The TSS is assumed to be free of design faults. It has to be certified at least to the same criticality level as the most critical host in the entire SoC.

**Replica determinism.** The TTSoC architecture supports different topologies with respect to the NoC which range from simple shared buses to complex mesh structures supporting multiple channels and concurrent message transfer. In advanced topologies the paths between different sender and receiver pairs can have different length (i.e. they can include a different number of hops), and a message on a short path can be received before a message that has been sent earlier, but over a longer path. Furthermore, multi-cast communication can be temporally asymmetric since also the paths from a single sender to the individual receivers may have different length. Therefore, messages are potentially received in an inconsistent order.

In the TTSoC architecture, replica determinism is established by exploiting the global time base in conjunction with time-triggered communication and computational schedules. Computational activities are triggered after the last message of a set of input messages has been received by all replicas of an FTU. This instant is a priori known due to the predefined time-triggered schedules. Thus, each replica wakes up at the same global tick, and operates on the same set of input messages. The messages in this set are treated as if they would have been received simultaneously, which means that a potential inconsistent reception order of the messages in this set is irrelevant.

**Temporal predictability.** The predefined message schedule of the NoC assures that each micro component can use its guaranteed reserved bandwidth independently of the communication activities of the other micro components. Furthermore, the concept of a *pulsed data stream* supports the reservation of a defined bandwidth within a periodically recurring interval. The recurring intervals are called the pulses of a pulsed data stream. This concept fits perfectly to TMR in a time-triggered system. A replica of a typical FTU will periodically read the replicated inputs, perform incoming voting, do the application specific processing on the voted input data and send its output value to the next FTU.

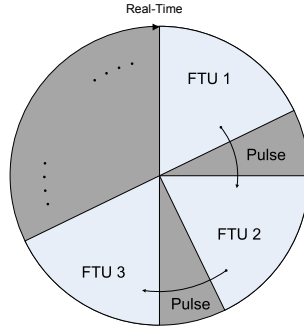


Figure 3. Voting on a Circular Time Model

The reactivity of the overall system can be optimized if the execution of the FTUs and the transmission of the messages are temporally aligned. Figure 3 depicts the execution of three FTUs in the periodic time model. The individual replicas of an FTU can execute in parallel since they reside on dedicated micro components. The pulses of the pulsed data streams are perfectly aligned with the execution of the replicas, and thus increase the reactivity of the system by minimizing the end-to-end latency from the first to the last FTU in the chain.

In addition to minimizing the end-to-end latency, the concept of a pulsed data stream increases the resource efficiency since the communication bandwidth is only reserved for those intervals in which it is actually needed.

**Recovery and Reintegration.** For on-chip TMR, the RCU – being a trusted component – coordinates the recovery actions of an FTU. Therefore, the messages of all replicas of an FTU are routed to the RCU where they are compared to each other. If one replica deviates from the other two replicas, the RCU sends to the TNA a *restart-request message* for the host on which the erroneous replica resides. Since the TNA has direct access to the TISSs and the TISSs control the reset lines of the attached hosts, the TNA can restart the corresponding host.

After the restart of a replica, the replica has to build up a valid internal state which has to be in perfect synchrony to state in the other replicas of the FTU. To facilitate state recovery, each replica periodically sends out its internal state via a so-called *history state message*. With the same period, each replica votes over the three history state messages (the own history state message and messages of the other two replicas) and overwrites its internal state with the voted result at the same global tick of its local clock. We call these periodic global ticks, the *reintegration points* of an FTU. Since the state of a replica after a reintegration point is exclusively determined by its inputs (data inputs plus the history state messages), a replica can be considered as stateless at the reintegration point. Therefore, a restarted component has simply to wait for the next reintegration point, to reach a consistent state.

If the replica has failed due to a transient fault, a restart of the corresponding host will be sufficient to reintegrate the replica. In order to detect a permanent or intermittent faults, the RCU can make use of threshold schemes like the  $\alpha$ -count [BCGG00]. Therefore, the RCU holds a failure counter for each replica, which is increased every time the replica deviates from the other replicas in an FCR, and which is decreased as time goes on. If the counter reaches a defined threshold, the host on which the replica resides is considered to be permanent faulty. In this case, the RCU can send a reconfiguration request to the TNA to remove the faulty host from the FTU

and replace it by a spare host. The spare host has to hold the images of all the replicas for which it should potentially act as a substitute.

## Off-chip TMR

To achieve ultra-high dependability, the TTSoC architecture supports the construction of FTUs where the individual replicas are mapped on the hosts of *distinct* SoCs which are interconnected by a fault-tolerant off-chip network like TTP [KG94], FlexRay [Fle05] or Time-triggered Ethernet [KAGS05] (see Figure 2(b)). Thus, the SoCs form network nodes of a fault tolerant distributed system. Since each replica of an FTU is located on a distinct network node, an FTU will still stay operational despite the failure of an entire node. We will now show how the required architectural properties for TMR are met with respect to off-chip TMR.

**Encapsulation.** In ultra-high dependable systems common mode failures have to be considered that can cause an entire chip to fail. Examples are disturbances in the power supply, particle induced multi-bit errors, extensive EMI disturbances, or physical damage of the chip. Thus, a single host in an SoC can no more be regarded as an FCR.

For off-chip TMR we consider an entire SoC as an FCR for physical faults. The coverage of the assumption that the nodes in a distributed system fail independently is much higher than for hosts within a single SoC. Contrary to host in an SoC, the network nodes of a distributed system do not reside on the same die nor in the same package. They can be physically separated over large distances (e.g., on the opposite sides of a car or an airplane), and can have individual power supplies.

The TTSoC architecture requires that faulty nodes cannot interfere with the correct operation of the off-chip network. Furthermore, the off-chip network has to be able to tolerate internal faults in order to meet the requirements for ultra-high dependability. Therefore, it has to be partitioned into multiple FCRs, which are integrated in a way that the network can deliver a correct communication service despite of a failure of a single internal FCR.

An example of an ultra-high dependable network that meets these requirements is TTP. TTP provides error containment via local or centralized bus guardians which electronically connect each node only during its specified time slot to the shared communication bus. Thus, a node which is violating its temporal specification cannot disrupt the communication among the other nodes. Furthermore, TTP was constructed to tolerate any arbitrary single fault within the network itself, by providing two redundant communication channels (with dedicated bus guardians) forming independent FCRs.

As for on-chip TMR, we consider a host as an FCR for design faults. If pieces of a design are used in a set hosts, the entire set has to be considered as a single atomic FCR. The off-chip network is considered to be free of design faults (e.g., TTP is certified for the usage in ultra-high dependable systems).

**Replica determinism.** As described in section 2, replica determinism in the TTSoC architecture is based on a consistent view of the global time. Time-triggered networks like FlexRay, TTP or TTE provide fault-tolerant clock synchronization to establish a system-wide global time. To facilitate the temporal coordination of hosts residing on different nodes, the *chip-wide* global time in each SoC is synchronized to the *system-wide* global time established by the off-chip network.

**Temporal predictability.** Temporal predictability is provided by the combination of the on-chip and the off-chip time-triggered network. Due to the fact that the time in the NoC is synchronized to the system-wide time of the off-chip network, the message schedule of both networks can be aligned which makes the exchange of messages at the on-chip/off-chip gateways temporally predictable.

**Recovery and Reintegration.** For off-chip TMR there is no central unit that can trigger a recovery action of replicas that are distributed across multiple SoCs. The restart or the migration of a replica can be exclusively triggered by the RCU within the SoC on which the replica resides. In order to enable recovery for off-chip TMR, the output messages off all three replicas of a distributed FTU have to be routed to the three RCUs of the SoCs where the replicas reside. By comparing the output messages, a RCU can decide whether the local replica functions correctly or whether it is affected by a transient or permanent fault. In case of transient fault the replica can be restarted, and in case of a permanent fault of the corresponding host, the replica can be migrated to a spare host.

Thus, from a global point of view, an SoC in a distributed system, is a self-checking component in which recovery actions are exclusively triggered by the local RCU. The limitation of this approach is, that the recovery actions can only be taken as long as the TSS in the SoC is still operational. The advantage is that a central coordination unit for recovery is not required, which would constitute a single point of failure and would not be acceptable for ultra-high dependable systems.

## 4. Reliability Modeling

This section describes the model and the evaluation approach for the quantitative reliability assessment of a single SoC, as well as for different TMR implementations. The model takes into account the increasing importance of transient faults by Single Event Upsetss (SEUs) due to shrinking semiconductor geometries and lower power voltages. In addition, the reliability assessments focus on the consequences of design faults in the context of replicated application computers and their design fault correlation.

For the evaluation of the TTSoC architecture, the Mobius dependability modeling tool [CCD<sup>+</sup>01] has been used. Mobius is a software tool for modeling the performance and dependability of complex systems. It uses an integrated multi-formalism, multi-solution approach, i.e., the overall model can be divided into smaller pieces and treated with different model formalisms and solution techniques. The classical modeling work is done in atomic and composed models. Atomic models are the basic modeling elements which consist of states, state transitions and parameters. Composed models are used to form more complex models by assembling atomic- or other composed models.

### Chip-Model

**Fault Model of the TTSoC.** The fault model for a SoC component is partitioned into a design fault model and a physical fault model. The design fault model comprises hardware and software design faults and the physical fault model describes faults caused by physical exposures. Because of the limited independence of IP cores within an SoC, the conventional approach for a fault hypothesis would be the consideration of the complete chip as a single FCR for physical- and design faults, as outlined in [Kop97]. In our case, justified by the distributed, fault-tolerant architecture of the TTSoC architecture, our fault hypothesis exhibits a more detailed structuring of FCRs. As

a result the probability for common mode failures is higher than the probability when considering a complete SoC as an FCR.

**Design fault model.** The design fault model contains the following FCRs:

- **Replicated Host:** This FCR includes the application computer and the front end. In case of replication without diversity, all replicas (possibly on different SoCs) comprise the same FCR.
- **TSS:** This FCR consists of the RCU, the NoC, the TNA and the TISSs. The TSS is assumed to be free of design faults. The simplicity of the TSS's design in the TTSoC architecture facilitates a thorough validation (e.g., by means of formal verification) which substantiates this assumption.
- **RMA:** The RMA forms its own FCR, because the RMA will usually be developed by different vendors than the TSS and exhibits a lower rigidity in the development process. Due to the high complexity of the RMA which has to dynamically compute time-triggered communication schedules from the reconfiguration requests of the application, validation to the highest criticality levels, e.g., class A according to DO-178B will generally be infeasible. The potential residue of design faults in the RMA is taken into account by means of a resource allocation protection by the TNA. The TNA protects safety-critical application subsystems from being disturbed by reconfiguration actions by a faulty RMA. Thus, only non safety-critical application subsystems can be affected by an RMA failure.

The corresponding failure rates can be estimated by means of the desired SILs (according to the IEC61508 standard) per FCR. When using design diversity each application computer can be considered as own FCR. The according assumption coverage is bounded by the design fault correlation which is determined by the usage of common resources like development tools, compiler, specification representation etc. ([AA88]). Figure 5(b) summarizes the design failure rates.

**Physical fault model.** The physical fault model contains the following FCRs:

- **Host:** For physical faults, a host in conjunction with the port manager constitutes an FCR. Neither the port manager nor the host can interfere with the correct operation of the pulse manager or the NoC.
- **Pulse managers + TNA + RCU + NoC:** The pulse manager is the only part of a micro component which has direct access to the NoC. So a failure of one of the pulse managers can cause an overall SoC failure. Also the TNA can cause a common mode failure and therefore it is reasonable to combine all pulse managers and the TNA together into one FCR.
- **RMA:** Since the resources of safety-critical application subsystems are protected by the TNA, a fault in the RMA can affect exclusively non safety-critical application subsystems.

We estimated the transient failure rates over the resource usage (ram, flip flops) of a TTSoC prototype implementation on an Altera Cyclone II Field Programmable Gate Array (FPGA). The method is described in [GA05] and the necessary SEU radiation tests can be found in [iT05]. Figure 5(a) summarizes the physical failure rates.



**Repair and recovery durations.** In the fault model we distinguish between repair and recovery in that way that repair removes permanent faults and is executed by an external force while recovery happens autonomously after transient faults. We got concrete values for the durations from typical values of the aviation and the automotive domain: For the Mean Time To Repair (MTTR) we took the typical airplane mission time of 10 hours and for the recovery time we took the maximum allowed activator freezing time of 50 ms for automotive systems [HT98].

**Mobius Model of the TTSoC.** To describe the TTSoC system two different model formalisms are combined: the Rep & Join formalism for an abstract (composed) view of the SoC and the SAN formalism to describe the sub-models in detail. Generally, the SoC model is a composition of one atomic SoC-infrastructure model (describes RMA, RCU, TNA), a gateway model (models the interconnection to an off-chip network) and an application-dependent number of micro component models. Within the composed model the atomic models share informations by means of shared state variables which distribute, e.g., the state of the TSS or the number of failed hosts etc..

**Atomic Model of SoC RMA, RCU and TNA.** A collective model describes the failures and recoveries of TNA, RMA and RCU. In case of a TNA failure the whole SoC is considered to be failed. When a RMA failure happens all non safety-critical application subsystems are considered to fail too. A RCU failure causes a failure of all safety-critical application subsystems which are part of the FTU.

**Atomic Model of a Micro Component.** The SAN model captures a replicated micro component used for TMR. For this purpose the model share a state variable to count the number of failed replicas and a state variable to set or reset the state of the assigned DAS.

The model covers the following events:

- Physical faults in the TISS–pulse manager: A physical fault in the pulse manager results in a NoC failure and therefore to an overall SoC failure. Therefore a TSS failure is communicated to all micro component model instances which logical belong to the same SoC.
- Physical faults in host and TISS–port manager: Cause a failure in the value domain and therefore only concern the micro component model.
- Application computer design faults: Determined by the coverage of the design diversity, a design fault causes either the failure of one or the simultaneous failure of all replicated micro components. Within the model this circumstance is modeled by a transition which possesses two cases, one with the probability of the design-fault-correlation and one with the probability 1 - design-fault-correlation.
- TSS failures caused by the RCU+RMA+TNA model or other micro component models: The model gets the TSS state by a state variable shared with all other models logical belonging to the same SoC.
- Gateway failures: The model gets the state of the off-chip interconnection by a state variable shared with the Gateway model.

## Model of Multiple SoCs with Gateways

For our distributed model the interconnection takes place over a fault tolerant star-coupled time-triggered network (e.g., TTP [KG94] or TTE [KAGS05]), see Figure 4. Switches are enclosed

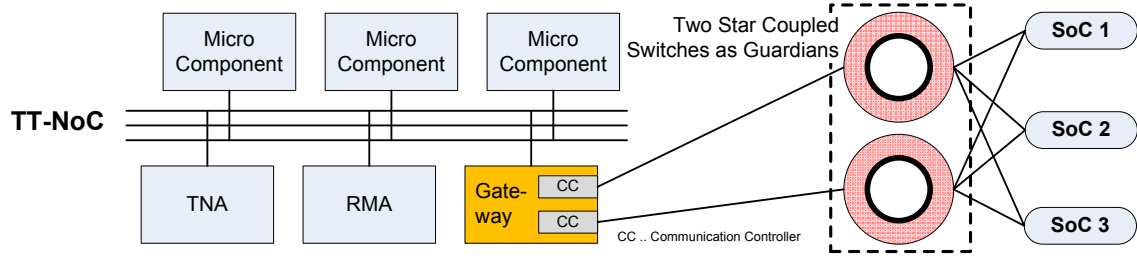


Figure 4. Fault tolerant SoC Cluster

from Guardians which protect the whole network from temporal domain switch failures. Each Guardian has exact knowledge about the sending times of each SoC and acts as temporal firewall in the same way as the TISS for the Network-on-Chip does. The connection between a SoC and the cluster network is made by the gateway micro component (see Figure 4). The gateway consists of a TISS, a gateway host as intelligent mediator between intra and inter-SoC network and two redundant communication controllers.

FCR	Transient Failure Rate	Permanent Failure Rate
Host	$10^4$ FIT	1 FIT
TISS-Port manager	170 FIT	0.02 FIT
TISS-Pulse manager	267 FIT	0.03 FIT
RMA	404 FIT	0.04 FIT
TNA	144 FIT	0.02 FIT
Switch	1000 FIT	0.1 FIT
Gateway Host	100 FIT	0.01 FIT
Communication Controller	700 FIT	0.07 FIT
Physical Link	$10^4$ FIT	1000 FIT

(a) Physical Fault Model

FCR	SIL	Failure Rate
Safety Critical Application Computer	4	10 FIT
Non Safety Critical Application Computer	1	1000 FIT
RMA	2	100 FIT
Trusted Subsystem	4	1 FIT

(b) Design Fault Model

Figure 5. Failure Rates for the Physical and Design Fault Model in FITs (failures in  $10^9$  hours)

The gateway fault model consists of the following FCRs:

- **Switch + Guardian:** Physically the Guardian is a part of the switch and in this way together they form a FCR.
- **Physical Link:** A physical link is a bidirectional physical connection between a SoC and a Switch. Because the two physical links to the redundant off-chip network are spatially close together the assumption coverage is bounded by the probability of correlated channel failures (e.g. caused by Electro Magnetic Interference (EMI)).
- **Communication Controller:** A communication controller is part of the gateway micro component.
- **Gateway Application Computer:** A gateway application computer has the purpose of mediation between the NoC and the off-chip network.

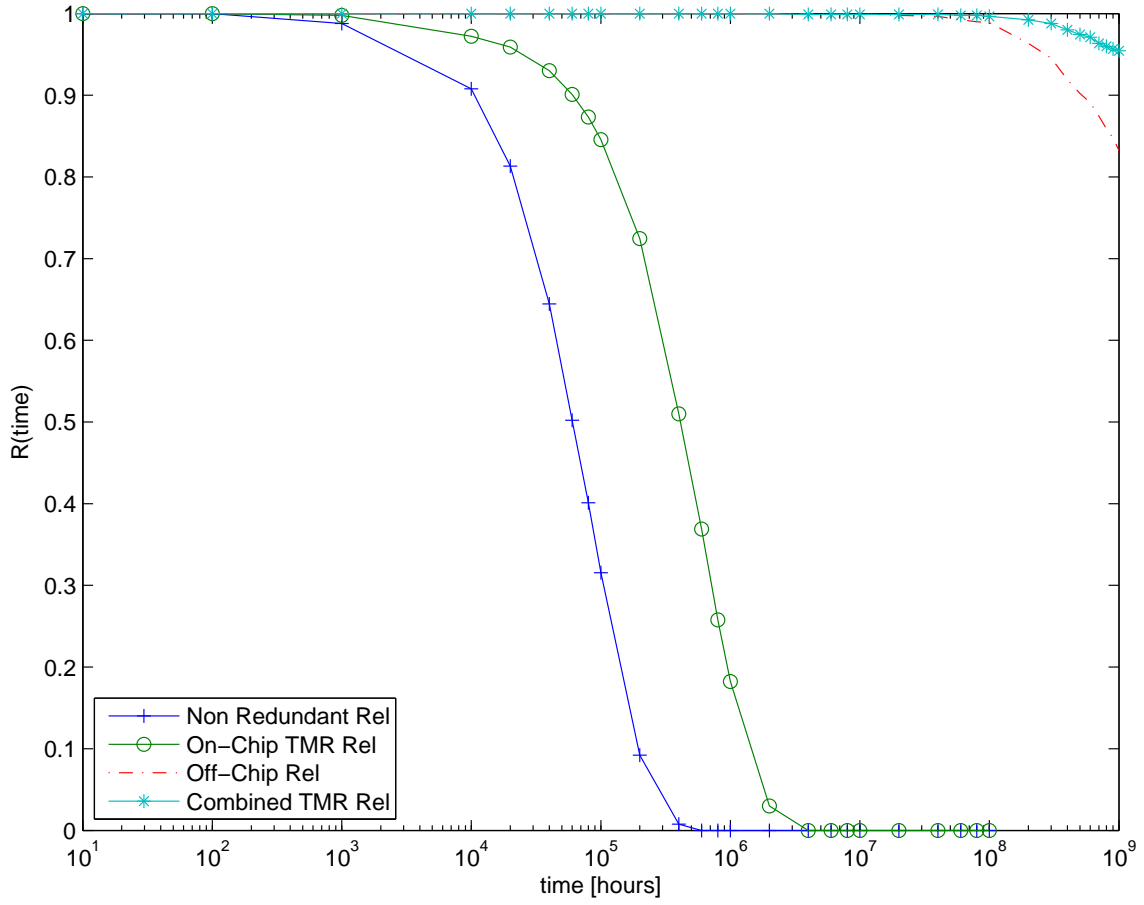


Figure 6. Reliabilities of TMR (application computer failure rate of 10000 FIT)

The failure rates for communication controller and gateway application computer were estimated over their resource usages as mentioned for the SoC fault model. The physical link failure rate was taken from typical EMI rates and the failure rate for the switch was taken from typical assumptions about electrical devices (see table 5(a)). For the repair and recovery durations the same assumptions as in the chip fault model were used.

## 5. Results

This section compares the simulation results of four different system configuration: non redundant, on-chip TMR, off-chip TMR, and combined on-chip and off-chip TMR. Figures 5(a) and 5(b) depict the model parameters, while Figures 7 and 6 show the resulting Mean Time To Failures (MTTFs) and reliabilities for the different TMR approaches.

### Comparison of On-chip, Off-Chip and Combined On-chip/Off-Chip TMR

As can be seen in Figures 7 and 6, on-chip TMR clearly outperforms a non-redundant solution for host failure rates of 500 FIT or worse. In case of host failure rates better than 500 FIT, the failure rate of the TSS is dominant and undermines potential reliability gains through active

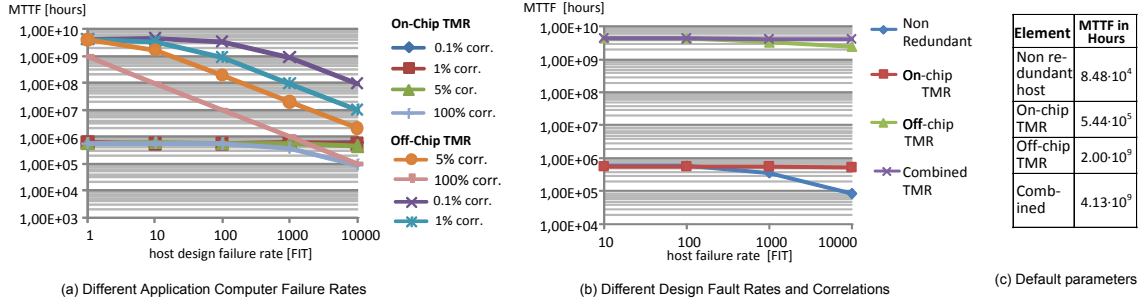


Figure 7. Comparison of MTTF in Different TMR Configurations

replication of hosts. Also, in case of low host failure rates compared to the failure rate of the TSS, on-chip TMR does not contribute towards a better reliability, because each additional TISSs worsens the reliability of the TSS. In Figures 7 and 6, even a degradation of reliability can be observed for host failure rates smaller than 100 FIT. Nevertheless, the failure rate of the TSS is assumed to be much lower than the failure rate of a host due to the smaller area consumption. This assumption has been confirmed by first prototype implementations of the TTSoC architecture.

Figures 7 and 6 also demonstrate that off-chip TMR is the basis for supporting ultra-dependable applications, i.e., reliability requirements in the order of 1 FIT. Without off-chip, the reliability of a cluster follows from the reliability of a single SoC and does not exceed 1000 FIT.

The performance of off-chip TMR can be increased by combining on-chip and off-chip TMR. This effect gains particular significance for high host failure rates, e.g., as can be expected from transient faults in future VDSM technology.

## Application Computer Design Failure Rate & Diversity Coverage

Figure 7(b) depicts the MTTF (caused by physical and design faults) of a system with one or more SoCs. Along the horizontal axis of Figure 7(b), different rates of failures caused by design faults are distinguished. The figure also shows the effect of different correlations between FCRs (between 0.1% and 100%). The rate of failures caused by physical faults is not varied.

For on-chip TMR, the reliability improvements of design diversity is less significant because of the dominant failure rates due to physical faults. For off-chip TMR, on the other hand, diversity has a significant positive impact for increasing failure rates due to design faults.

## 6. Conclusion

Future MPSoCs will have to cope with the increasing transient failure rates induced by VDSM technology. This paper has presented a solution for addressing this challenge based on TMR of IP cores in a novel MPSoC architecture with a time-triggered on-chip network. Compared to previous approaches for mitigating transient faults, the presented solution offers superior time and energy efficiency, enables the use of standard IP core libraries, improves resilience against spatial proximity faults, and establishes the foundation for design diversity and heterogeneity. Key mechanisms for TMR of IP cores are the inherent fault isolation and the determinism of the time-triggered on-chip network. The latter property (also known as replica determinism) ensures that correct replicas always reach the same computational result within a bounded time interval, which is required for exact voting. The former property is important for preserving the independence of replicas by preventing common mode failures of replicas.

Another focus of the paper has been the combination of on-chip TMR with off-chip TMR. Since individual MPSoCs cannot be expected to achieve a reliability as required for ultra-dependable systems (i.e., in the order of  $10^{-9}$  failures/hour), active redundancy employing multiple MPSoCs is necessary.

Both the benefits of on-chip TMR, as well as the combination of on-chip and off-chip TMR have been analyzed quantitatively using reliability modeling. In particular, the results have demonstrated that on-chip TMR contributes significantly towards improving reliability of MPSoCs in the presence of high host failure rates caused by transient faults in VDSM technology.

## 7. Acknowledgments

This work has been supported in part by the European IST project ARTIST2 under project No. IST-004527 and the European IST project DECOS under project No. IST-511764.

## References

- [AA88] W. Schutz A. Avizienis, M.R. Lyu. In search of effective diversity: a six-language study of fault-tolerant flight control software. *Eighteenth International Symposium on Fault-Tolerant Computing (FTCS-18)*, 1988.
- [AAN00] L. Anghel, D. Alexandrescu, and M. Nicolaidis. Evaluation of a soft error tolerance technique based on time and/or space redundancy. In *SBCCI '00: Proceedings of the 13th symposium on Integrated circuits and systems design*, page 237, Washington, DC, USA, 2000. IEEE Computer Society.
- [AN00] L. Angheland and M. Nicolaidis. Cost reduction and evaluation of a temporary faults detecting technique. In *Proc. of Design, Automation and Test in Europe Conference and Exhibition*, pages 591–598, mar 2000.
- [BCGG00] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, and F. Grandoni. Threshold-based mechanisms to discriminate transient from intermittent faults. *Transactions on Computers*, Vol.49, Iss.3, Mar 2000, 49:230–245, 2000.
- [CCD<sup>+</sup>01] "G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J.M. Doyle, W.H. Sanders, and P. Webster". "the mobius modeling tool". In *"Proc. of 9th International Workshop on Petri Nets and Performance Models"*, pages 241–250, sept 2001.
- [Con02] C. Constantinescu. Impact of deep submicron technology on dependability of VLSI circuits. In *Proc. of the Int. Conference on Dependable Systems and Networks*, pages 205–209. IEEE, 2002.
- [Con03] C. Constantinescu. Trends and challenges in vlsi circuit reliability. *IEEE Micro*, 23(4):14–19, 2003.
- [DNR02] E. Dupont, M. Nicolaidis, and P. Rohr. Embedded robustness IPs for transient-error-free ICs. In *Proc. of the Conference on Design, Automation and Test in Europe*, page 244, Washington, DC, USA, 2002. IEEE Computer Society.
- [Fle05] FlexRay Consortium. BMW AG, DaimlerChrysler AG, General Motors Corporation, Freescale GmbH, Philips GmbH, Robert Bosch GmbH, and Volkswagen AG. *FlexRay Communications System Protocol Specification Version 2.1*, May 2005.
- [GA05] M.B. Tahoori G. Asadi. Soft Error Rate Estimation and Mitigation for SRAM-Based FPGAs. *International Symposium on Field Programmable Gate Arrays*, 2005.
- [Gai06] J. Gaisler. The leon3ft-rtax processor family and seu test results. In *Proc. of the 9th Annual Military and Aerospace Programmable Logic Devices International Conference*, Washington, D.C., September 2006. Gaisler Research.
- [GAM<sup>+</sup>02] P. Gil, J. Arlat, H. Madeira, Y. Crouzet, T. Jarboui, K. Kanoun, T. Marteau, J. Duraes, M. Vieira, D. Gil, J.C. Baraza, and J. Gracia. *Fault Representativeness*. LAAS-CNRS, Toulouse, France, 2002. Deliverable (ETIE2) of the European Project Dependability Benchmarking Dbench (IST-2000-25425).
- [Gel01] P. Gelsinger. Microprocessors for the new millenium, challenges, opportunities, and new frontiers. In *Proc. of the Solid State Circuit Conference*. IEEE Press, 2001.
- [GSVP03] M.A. Gomaa, C. Scarbrough, T.N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multi-processors. *IEEE Micro*, 23(6):76–83, 2003.

- [H.K06] H.Kopetz. Pulsed data streams. In *IFIP TC 10 Working Conference on Distributed and Parallel Embedded Systems (DIPES 2006)*, pages 105–124, Braga, Portugal, October 2006. Springer.
- [HT98] G. Heiner and T. Thurner. Time-triggered architecture for safety-related distributed real-time systems in transportation systems. In *Proc. of the Twenty-Eighth Annual Int. Symposium on Fault-Tolerant Computing*, pages 402–407, June 1998.
- [Int94] Int. Standardization Organisation, ISO 7498. *Open System Interconnection Model*, 1994.
- [iT05] iRoC Technologies. Radiation results of the ser test of actel fpga december 2005. Technical report, December 2005.
- [JR94] Lala J.H. and Harper R.E. Architectural principles for safety-critical real-time applications. In *Proceedings of the IEEE*, volume 82, pages 25–40, jan 1994.
- [KAGS05] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The Time-Triggered Ethernet (TTE) design. *Proc. of 8th IEEE Int. Symposium on Object-oriented Real-time distributed Computing (ISORC)*, May 2005.
- [KG94] H. Kopetz and G. Grunsteidl. TTP – a protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, January 1994. Vienna University of Technology, Real-Time Systems Group.
- [KN97] H. Kopetz and R. Nossal. Temporal firewalls in large distributed real-time systems. *Proc. of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '97)*, 1997.
- [Kop92] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proc. of 12th Int. Conference on Distributed Computing Systems*, Japan, June 1992.
- [Kop97] H. Kopetz. *Real-Time Systems Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, 1997.
- [LV62] R.E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200, April 1962.
- [MPG02] S. Mishra, M. Pecht, and D.L. Goodman. In-situ sensors for product reliability monitoring. In *Proc. of SPIE*, volume 4755, pages 10–19, 2002.
- [Nic99] M. Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *Proc. of the 17th IEEE VLSI Test Symposium*, pages 86–94, April 1999.
- [OMG02] OMG. Smart Transducers Interface. Specification ptc/2002-05-01, Object Management Group, May 2002. Available at <http://www.omg.org/>.
- [PMH98] B. Pauli, A. Meyna, and P. Heitmann. Reliability of electronic components and control units in motor vehicle applications. In *VDI Berichte 1415, Electronic Systems for Vehicles*, pages 1009–1024. Verein Deutscher Ingenieure, 1998.
- [Pol94] S. Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6:289–316, 1994.
- [Pol96] S. Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, 1996.
- [Sem06] Semiconductor Industry Association (SIA). International technology roadmap for semiconductors – 2006 update. Technical report, 2006.
- [SKK<sup>+</sup>02] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389–398, Washington, DC, USA, 2002. IEEE Computer Society.
- [SWH95] N. Suri, C.J. Walter, and M.M. Hugue. *Advances In Ultra-Dependable Distributed Systems*, chapter 1. IEEE Computer Society Press, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA 90720-1264, 1995.