Institut für Automatisierungs- und Softwaretechnik

Universität Stuttgart

Pfaffenwaldring 47

D-70569 Stuttgart

Diplomarbeit Nr. 2352

Conception and Implementation of an Agreement Protocol for Fault-Tolerant Automotive Embedded Systems

Mourad Limam

Studiengang:	Informatik
Prüfer:	Prof. Dr.-Ing. Dr. h. c. Peter Göhner
Betreuer:	Dipl.-Ing. Michael Wedel
begonnen am:	24.05.2005
beendet am:	24.11.2005

CR-Klassifikation: C.2.4 C.3 C.4

Experiences and Acknowledgements

As a Computer Science student, this diploma thesis (Diplomarbeit) allowed me to widen my scope by exploring the highly interesting field of automotive electronics. Besides applying my knowledge in distributed systems and real-time programming, I also learned about automotive electronic systems, modern automotive communication systems and fault-tolerance concepts in safety-relevant electronic systems.

Since this work has been conducted in partnership with the DaimlerChrysler AG, I had the opportunity to learn to use the prevalent development tools in the automotive industry, in particular for the development of FlexRay applications. I also worked close to several other projects in the company, which provided me with profound insights into the methods and processes for the realization of industrial projects.

Moreover, this work has been conducted according to the process model of the Institute of Industrial Automation and Software Engineering, which specifies that a student has to commit to stringent requirements concerning the documentation created and several deadlines representing the project milestones. This appears to be a lot of work initially. I believe, however, that this process model has only benefits as it allowed me to formulate a clear idea about what was to be done, very early in the project, allowing me to create a project plan before work began. Furthermore, the creation of documents during the project allowed not only my supervisors, but also me to assess my own understanding of the relevant aspects and significantly reduced the effort and time for writing the final thesis paper.

During my project, I had the opportunity to have many people around me at the IAS, as well as at DaimlerChrysler AG, whom I could ask for advice anytime I needed it and who contributed significantly to the successful completion of this work. Therefore, I would like to express my deepest gratitude to my supervisor Michael Wedel from the IAS and to my supervisors Dr. Bernd Hedenetz and Mr. Xi Chen from DaimlerChrysler AG.

Table of Contents

EXPERIENCES AND ACKNOWLEDGEMENTS II

TABLE OF CONTENTS III

LIST OF FIGURES VII

LIST OF TABLES X

LIST OF ABBREVIATIONS XI

TERMINOLOGY XIII

ZUSAMMENFASSUNG XV

ABSTRACT XVI

1 INTRODUCTION 1

1.1 MOTIVATION 1

1.2 OVERVIEW OF THE THESIS 2

2 AUTOMOTIVE SAFETY-RELEVANT ELECTRONICS 4

2.1 ELECTRONIC SYSTEMS IN THE AUTOMOTIVE INDUSTRY 4

2.1.1 Components of an Automotive Electronic System 4

2.1.2 Functions and Complexity 6

2.2 SAFETY-RELEVANT AUTOMOTIVE ELECTRONIC SYSTEMS 7

2.2.1 Requirements of Fault-Tolerance 7

2.2.2 Technical Requirements for Safety-Relevant Electronic Systems 7

2.3 DEVELOPMENT PROCESSES FOR SAFETY-RELEVANT ELECTRONIC SYSTEMS 8

2.3.1 The V-Model 8

2.3.2 Safety Relevant Development Steps 9

2.3.3 Model-Based Design using the Example of Simulink 10

2.4 A STANDARD SOFTWARE ARCHITECTURE FOR INTEGRATED SAFETY APPLICATIONS 12

2.4.1 The Motivation for Standardization 12

2.4.2 EAST-EEA 13

2.4.3 EASIS 14

3 ASPECTS OF FAULT-TOLERANCE IN SAFETY-RELEVANT ELECTRONIC SYSTEMS 15

3.1 BASICS OF FAULT-TOLERANCE 15

3.1.1 Failure, Error and Fault 15
3.1.2 Fault-Tolerance, Reliability and Safety 16
3.1.3 Structural Redundancies and Non-Determinism 17
3.1.4 Fault Masking for Structural Redundancy 18
3.2 FAULT-TOLERANT ARCHITECTURES 20
3.2.1 Duplex Systems 20
3.2.2 Fault-Masking Dual Duplex Systems 21
3.2.3 Fault-Masking Triplex Systems 21
4 TIME-TRIGGERED ARCHITECTURE FOR SAFETY-RELEVANT ELECTRONIC SYSTEMS 22
4.1 TIME-TRIGGERED VS. EVENT-TRIGGERED SYSTEMS 22
4.2 TIME-TRIGGERED OPERATING SYSTEMS 23
4.3 TIME-TRIGGERED COMMUNICATION SYSTEMS USING THE EXAMPLE OF FLEXRAY 24
4.3.1 Architecture of a FlexRay Node 24
4.3.2 Network Topologies 25
4.3.3 FlexRay Communication Protocol 25
4.3.4 FlexRay Frame Format 27
4.3.5 Support for Reliability and Dependability Services 28
5 AGREEMENT PROTOCOLS FOR SAFETY-RELEVANT ELECTRONICS 29
5.1 ORAL MESSAGES ALGORITHM 29
5.2 SIGNED MESSAGES ALGORITHM 31
5.3 PENDULUM PROTOCOL 32
5.4 COMPARISON AND CONCLUSION 34
6 CONCEPT FOR AN AGREEMENT PROTOCOL SERVICE 36
6.1 REQUIREMENTS AND CONSTRAINTS OF THE AP SERVICE 36
6.1.1 Requirements to the AP Service 36
6.1.2 Requirements of the AP Service 37
6.1.3 Constraints 37
6.2 INTEGRATION IN THE EASIS ARCHITECTURE 38
6.3 INTERNAL DESIGN OF THE AP SERVICE 39
6.3.1 The Signing Unit 41
6.3.2 The Decision Unit 42
6.3.3 The Masking Unit 43
6.3.4 The Impact of Communication Failures 46

6.4 EXAMPLES 47

7 IMPLEMENTATION OF A VIRTUAL PROTOTYPE 50

7.1 REQUIREMENT AND CONSTRAINTS 50

7.2 IMPLEMENTATION TOOLS 51

7.2.1 Matlab/Simulink Standard Blocks 51

7.2.2 S-Functions 52

7.2.3 Tool chain of DECOMSYS 53

7.3 IMPLEMENTATION 56

7.3.1 General Overview 56

7.3.2 Implementation of AP Frames 58

7.3.3 Implementation of the Signing Unit 59

7.3.4 Implementation of the Decision Unit 61

7.3.5 Implementation of the Masking Unit 61

7.3.6 Scheduling of SIMSYSTEM Tasks 65

7.3.7 Generation of the Communication Schedule 65

8 EVALUATION 68

8.1 EVALUATION CASE 1 68

8.1.1 Evaluation Procedure and Results 68

8.1.2 Evaluation of the Results 69

8.2 EVALUATION CASE 2 69

8.2.1 Evaluation Procedure and Results 69

8.2.2 Evaluation of the Results 70

8.3 EVALUATION CASE 3 70

8.3.1 Evaluation Procedure and Results 71

8.3.2 Evaluation of the Results 71

8.4 EVALUATION CASE 4 71

8.4.1 Evaluation Procedure and Results 72

8.4.2 Evaluation of the Results 72

8.5 EVALUATION CASE 5 72

8.5.1 Evaluation Procedure and Results 72

8.5.2 Evaluation of the Results 73

8.6 EVALUATION CASE 6 73

8.6.1 Evaluation Procedure and Results 73

8.6.2 Evaluation of the Results 74

8.7 EVALUATION CASE 7 74

8.7.1 Evaluation Procedure and Results 75
8.7.2 Evaluation of the Results 75
8.8 EVALUATION CASE 8 75
8.8.1 Evaluation Procedure and Results 76
8.8.2 Evaluation of the Results 76
8.9 CONCLUSION 76
9 SUMMARY AND OUTLOOK 77
9.1 SUMMARY 77
9.2 OUTLOOK 78
LITERATURE I
APPENDIX A – COMMUNICATION OVERHEAD OF THE OM AND SM ALGORITHMS III

List of Figures

Figure 1-1 Steer-by-Wire in the F500 Mind Research Vehicle 1
Figure 2-1 77 ECUs in the Maybach (luxury class) [Easi05a] 6
Figure 2-2 V-Model 9
Figure 2-3 Double V-Model 9
Figure 2-4 Simulink by “The MathWorks” [Math05a] 11
Figure 2-5 EAST architecture [IrJu04] 13
Figure 2-6 EASIS software architecture 14
Figure 3-1 A 2-out-of-3 System [Echt90] 18
Figure 3-2 A duplex system 20
Figure 3-3 A dual duplex system [EcBe02] 21
Figure 3-4 A triplex system [EcBe02] 21
Figure 4-1 FlexRay node architecture 24
Figure 4-2 FlexRay topologies [HSB+02] 25
Figure 4-3 FlexRay communication cycle [HSB+02] 26
Figure 4-4 FlexRay Frame Format [Flex02] 27
Figure 4-5 Error detection in FlexRay [HSB+02] 28
Figure 5-1 OM Algorithm with $n = 4$ and $m = 1$ 30
Figure 5-2 Distribution Protocol with 3 Nodes 32
Figure 5-3 Main Protocol (pendulum protocol) for 3 nodes, $\delta = 2$ 33
Figure 6-1 Interface to FTCom 38
Figure 6-2 Interface of the AP service 39
Figure 6-3 Different units constituting the AP service 40
Figure 6-4 AP Frame 40
Figure 6-5 Signing Unit 41
Figure 6-6 Decision Unit 43
Figure 6-7 Masking Unit 43
Figure 6-8 Example of the AP service execution 48
Figure 6-9 Simultaneous use of agreed values 48
Figure 6-10 Computation of a single agreed value 49

Figure 7-1 Constant Block 51
Figure 7-2 Mux and Demux Blocks 51
Figure 7-3 Goto and From Blocks 52
Figure 7-4 In and Out Blocks 52
Figure 7-5 Display Block 52
Figure 7-6 Cluster Block 53
Figure 7-7 Host Block 54
Figure 7-8 Task Block 54
Figure 7-9 Dispatch Event Block 54
Figure 7-10 Signal Read/Write Connectors 55
Figure 7-11 AAFM/VP Block 55
Figure 7-12 Scheduling Plug-In of DESIGNER 56
Figure 7-13 Modeling Steps 56
Figure 7-14 Prototype architecture 57
Figure 7-15 Model of a Node 57
Figure 7-16 Input Values and Parameters from Application 58
Figure 7-17 AP Service 58
Figure 7-18 S-Function Block: signFrames 59
Figure 7-19 S-Function Block: checkSignatures 60
Figure 7-20 S-Function Block: intervalDec 61
Figure 7-21 S-Function Block: genFrame 61
Figure 7-22 S-Function Block: appendID 62
Figure 7-23 S-Function Block: updateCV 62
Figure 7-24 S-Function Block: computeSV 63
Figure 7-25 First Communication Phase 63
Figure 7-26 Standard Communication Phase 64
Figure 7-27 Final Task 64
Figure 7-28 Scheduling of Tasks in a Node 65
Figure 7-29 Scheduling of LLIO Tasks 66
Figure 7-30 Communication Schedule 66
Figure 8-1 Prototype Execution for Case 1 69
Figure 8-2 Emulation of a Replication Fault in Node 1 70

Figure 8-3 Prototype Execution for Case 2 70
Figure 8-4 Prototype Execution for Case 3 71
Figure 8-5 Prototype Execution for Case 4 72
Figure 8-6 Prototype Execution for Case 5 73
Figure 8-7 Prototype Execution for Case 6 74
Figure 8-8 Prototype Execution for Case 7 75
Figure 8-9 Prototype Execution for Case 8 76
Figure 9-1 MicroAutoBox from dSpace [Dspa05] 78

List of Tables

Table 2-1 Bus Systems [Vect05] 5

Table 3-1 Masking decisions [Echt90] 19

Table 5-1 Criteria Priorities 34

Table 5-2 Comparison of three Agreement Protocols 35

Table 8-1 Evaluation Case 1 68

Table 8-2 Evaluation Case 2: One Faulty Node 69

Table 8-3 Evaluation Case 3: Non-Determinism, Tolerance = 0 71

Table 8-4 Evaluation Case 4: Non-Determinism, Tolerance = 1 71

Table 8-5 Evaluation Case 5: 1 Faulty Node, Non-Determinism, Tolerance = 1 72

Table 8-6 Evaluation Case 6: Communication Fault (1st Com. Phase) 73

Table 8-7 Evaluation Case 7: Communication Fault (2nd Com. Phase) 74

Table 8-8 Evaluation Case 8: 4-Node Configuration 75

List of Abbreviations

AAFM	**A**rchitecture **A**llocated **F**unctional **M**odel
ABS	**A**ntilock **B**rake **S**ystem
ADL	**A**rchitecture **D**escription **L**anguage
AP	**A**greement **P**rotocol
API	**A**pplication **P**rogramming **I**nterface
ASR	"**A**nti-**S**chlupf-**R**egelung", Anti-Slip-Control System
CAN	**C**ontroller **A**rea **N**etwork
COM	**COM**munication
CPU	**C**entral **P**rocessing **U**nit
CRC	**C**yclic **R**edundancy **C**heck
dll	**D**ynamically **L**inked **L**ibrary
EASIS	**E**lectronic **A**rchitecture **S**ystem Engineering for **I**ntegrated **S**afety Systems
EAST-EEA	**E**lectronic **A**rchitecture and **S**oftware **T**echnology – **E**lectronic **E**mbedded **A**rchitecture
ECU	**E**lectronic **C**ontrol **U**nit
EEPROM	**E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory
EPROM	**E**rasable **P**rogrammable **R**ead-**O**nly **M**emory
ESL	**E**lectronic **S**ystem **L**evel
ESP	**E**lectronic **S**tability **P**rogram
EU	**E**uropean **U**nion
FMEA	**F**ailure **M**ode and **E**ffect **A**nalysis
FTCom	**F**ault-**T**olerant **Com**munication
FTDMA	**F**lexible **T**ime **D**ivision **M**ultiple **A**ccess
I/O	**I**nput/**O**utput
ICV	**I**nteractive **C**onsistency **V**ector
ISS	**I**ntegrated **S**afety **S**ystems
IT	**I**nformation **T**echnology
LIN	**L**ocal **I**nterconnect **N**etwork
LLIO	**L**ow **L**evel **In** **O**ut

MOST	**M**edia **O**riented **S**ystems **T**ransport
NM	**N**etwork **M**anagement
OEM	**O**riginal **E**quipment **M**anufacturer
OM	**O**ral **M**essages
OS	**O**perating **S**ystem
OSEK	„**O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik im **K**raftfahrzeug“, Open Systems and the Corresponding Interfaces for Automotive Electronics
OSEKtime	An OSEK specification for a Time-Triggered Operating System
RAM	**R**andom **A**ccess **M**emory
ROI	**R**eturn **O**n **I**nvestment
ROM	**R**ead-**O**nly **M**emory
SM	**S**igned **M**essages
SV	**S**tate **V**ector
TDMA	**T**ime **D**ivision **M**ultiple **A**ccess
TTCAN	**T**ime **T**riggered **CAN**
TTP	**T**ime **T**riggered **P**rotocol
µC	**M**icro **C**ontroller
UML	**U**nified **M**odeling **L**anguage
V&V	**V**erification and **V**alidation
VP	**V**irtual **P**rototype
WCET	**W**orst **C**ase **E**xecution **T**ime
XML	e**X**tensible **M**arkup **L**anguage
XOR	e**X**clusive **OR**

Terminology

Agreement Protocol	A protocol-based fault masking algorithm executed among redundant components to agree on a common value (e.g. system state, sensor value, etc.) despite faults and non-determinism in the value domain.
Development Process Model	A systematic approach for development, which describes the different development phases that have to be followed in order to guarantee a high quality product.
Electronic Architecture and System Engineering for Integrated Safety Systems	An EU project launched in 2004 by a consortium consisting of OEMs, suppliers and software development companies. The project focuses on the conception of a standardized approach to the development of safety-relevant systems.
Error	An error arises in a system when it reaches a system state that was not anticipated during system design.
Fault-tolerance	The ability of a system to handle internal errors and faults in such a way that its failure is avoided or at least less severe with regard to its consequences.
FlexRay	A fault-tolerant communication system developed by the automotive industry, which is based on a protocol combining the time-triggered and the event triggered approach.
Failure	A behavior of a system, which deviates from the system specification
Fault	The technical cause for an error (see Error). It can be physical or algorithmic.
Fault Masking	A structural redundancy technique that completely masks faults within a set of redundant modules.
Model-based Development	A systematic design approach aiming at the early validation of the specification and design of the system to be developed by the means of a virtual prototype. A state-of-the-art development environment to produce virtual prototypes in the automotive industry is Matlab/Simulink.
Oral Messages Protocol	A variant of agreement protocols, based on the exchange of messages, which are not signed, between the nodes participating in the protocol.
Pendulum Protocol	A variant of agreement protocols, based on the exchange of signed

	messages between the nodes. This protocol executes in a similar way to a pendulum stroke over the nodes.
Safety-Relevant Systems	Electronic systems that have a safety function or whose malfunction could affect the safety of the vehicle and its occupants directly or indirectly.
Schedule	A time table that describes the execution scheme of tasks within a time-triggered operating system or the transmission scheme of communication frames within a time-triggered communication system
Signed Messages Protocol	A variant of agreement protocols, based on the exchange of signed messages between the nodes participating in the protocol.
Structural Redundancy	A fault-tolerance approach widely used in the aerospace industry and recently introduced into the automotive industry. It consists of the replication of system components to avoid single points of failure.

Zusammenfassung

Sicherheitsrelevante Elektroniksysteme im Fahrzeug haben besonders hohe Anforderungen an die Fehlertoleranz, vor allem wenn sie über keine mechanische Rückfallebene verfügen, wie zum Beispiel im Fall eines X-by-Wire Systems. Die Replikation von Komponenten, auch strukturelle Redundanz genannt, ist hierfür eine Lösung, die sich bewährt hat, um diese Systeme fehlertolerant zu gestalten. Die strukturelle Redundanz bringt allerdings auch unerwünschte Nebeneffekte mit sich, welche die Fehlermaskierung erschweren. Übereinstimmungsprotokolle sind protokollbasierte verteilte Algorithmen, die dazu eingesetzt werden, diese Effekte zu beseitigen und somit das Potenzial der strukturellen Redundanzen optimal auszunutzen. Diese Protokolle haben zum Ziel, eine Übereinstimmung unter verschiedenen Knoten bezüglich einer bestimmten Größe (z.B. Sensorwerte) durch einen organisierten Nachrichtenaustausch zu erreichen.

Das EU-Projekt EASIS strebt die Konzeption einer Standardsoftwareplattform für integrierte Sicherheitsapplikationen im Kraftfahrzeug an. Weiterhin verfolgt das DFG Projekt „Systemzuverlässigkeit in frühen Entwicklungsphasen“ die Erhöhung der Zuverlässigkeit mechatronischer Systeme unter anderem durch die Anwendung von Fehlertoleranzmaßnahmen. Der Schwerpunkt dieser Diplomarbeit liegt ausgehend davon auf der Konzeption und Implementierung eines Übereinstimmungsprotokolls als Standardsoftwaremodul, das in die EASIS Softwareplattform integriert werden soll. Verschiedene Ansätze zur Implementierung von Übereinstimmungsprotokollen in der Literatur wurden deshalb analysiert und verglichen. Basierend auf diesen Varianten wurde schließlich ein Konzept für ein Übereinstimmungsprotokoll für fehlertolerante sicherheitsrelevante Systeme erstellt. Das Konzept berücksichtigt nicht nur die Anforderungen an sicherheitsrelevante Elektroniksysteme, sondern auch den Standardisierungsaspekt im Rahmen des EASIS Projekts.

In Anlehnung an den modellbasierten Ansatz, wurde ein virtueller Prototyp in Matlab/Simulink auf Basis einer simulierten zeitgesteuerten Architektur entwickelt. Dieser Prototyp stellt ein Modell für ein System dar, das Redundanzen beinhaltet und in dem der implementierte Übereinstimmungsprotokolldienst validiert werden konnte. Die Simulationsergebnisse zeigten die korrekte Funktionalität des Übereinstimmungsprotokolldienstes und validierten damit das vorgestellte Konzept.

Abstract

Safety-relevant automotive systems have particularly high requirements for fault-tolerance, especially in the absence of a mechanical backup, such as for X-by-Wire systems. The replication of components, called structural redundancy, is very often a way to ensure that these systems are free from single points of failure and, hence, fault-tolerant. However, the use of redundancies also implies undesirable effects which make the masking out of faults difficult. Agreement protocols are protocol-based, distributed algorithms which are required to eliminate these effects, in order to profit from the potential of redundancies optimally. These protocols aim to establish agreement among different nodes with respect to a particular value (e.g. sensor value) through an organized message exchange.

As part of the EU project EASIS, which aims to provide a standard software platform for integrated safety applications, and the DFG Project “System Reliability”, this diploma thesis (Diplomarbeit) focuses on the conception and implementation of an agreement protocol as a standard software module to be integrated in the EASIS software platform. Therefore, several approaches to implementing agreement protocols from the literature are analyzed and compared. Based on these variants, a concept for an agreement protocol for fault-tolerant safety electronics is finally presented, where not only the specific requirements for safety electronic systems are taken into account, but also the main stream of standardization, particularly in the scope of the EU Project EASIS.

Following the model-based design, a prevalent approach for the design of electronic systems in the automotive industry, a virtual prototype has been developed in Matlab/Simulink, based on the time-triggered paradigm. This prototype represents a system model including redundant components, where the implemented Agreement Protocol service can be validated. The simulation results for several evaluation cases asserted the functionality of the Agreement Protocol service and, thus, validated the suggested concept.

1 Introduction

Safety electronic systems in modern vehicles can generally be subdivided into two categories: active safety systems and passive safety systems. While active safety systems, such as the ABS system, help to prevent accidents by assisting the driver in crucial situations, passive safety systems, such as the airbag system, aim to reduce the negative impact of an accident (e.g. personal injury) by absorbing the resulting crash energy. From a technical point of view, these systems are mostly stand alone solutions with a limited degree of interdependency. In order to reach the road safety goals set by the European Commission Transport Policy for 2010, these systems must be integrated into a complete network of so-called Integrated Safety Systems, whose realization requires a powerful and highly dependable in-vehicle electronic architecture, as well as appropriate development support. These elements must be standardized to achieve an improvement in system quality with shorter development times and lower system costs. The goal of the EASIS project is to define and develop these enabling technologies [Easi05c].

As of the standardization of the software architecture, a standard software platform, mainly consisting of standard dependability software modules, such as a Fault-Tolerant Communication service and an Agreement Protocol service, will be developed in the scope of the EASIS project. As part of this project, this diploma thesis (Diplomarbeit) focuses particularly on the conception and development of the Agreement Protocol service, which aims to support the use of redundancies in fault-tolerant electronic systems, and its integration into the EASIS software platform as a standard software module.

1.1 Motivation

While room for innovation in the field of passive safety systems has become very slim, active safety systems still offer great potential for further improvement. The trend is even moving towards the substitution of conventional mechanical systems by mechatronic ones, which would allow the realization of new safety and comfort features. However, in the absence of a mechanical backup, such as in the case of X-by-Wire systems (see Figure 1-1), the requirements of dependability and, thus, of fault-tolerance for these systems are particularly high.

Figure 1-1 Steer-by-Wire in the F500 Mind Research Vehicle

One way to fulfill these stringent requirements for fault-tolerance in safety-relevant electronics is the use of structural redundancies, i.e. the replication of system components to avoid single points of failure. Further mechanisms are then required in order to mask out the behavior of faulty components in the system. However, due to undesirable effects, such as slight deviations in redundant sensor values, called non-determinism in the value domain, the masking out of faults in a fault-tolerant architecture becomes difficult.

The problem of non-determinism in the value domain can be only solved through cooperation between redundant components in order to agree on a common value. However, in the presence of faulty components that might dissipate faulty and inconsistent information, the problem becomes even more difficult. By describing a specific communication scheme for the exchange of values between the redundant components, agreement protocols help to overcome these difficulties, so that agreement among the non-faulty system components can be reached. Therefore, their use is inevitable in safety-related electronics to exploit the potential of redundancies fully.

1.2 Overview of the Thesis

The field of electronics in the automotive industry has acquired a great importance over the past years. It is currently estimated that 90% of the innovations in vehicles are realized in electronics. Section 2 gives a general overview of automotive electronics and focuses mainly on the safety-relevant electronics, by presenting their requirements and the development processes adopted for these systems, as well as a standard software architecture for the development of integrated safety applications.

In section 3, the relevant concepts for fault-tolerance in safety-relevant electronics are described. Depending on the level of fault-tolerance to be achieved, different architectures including structural redundancies are also presented. The replication of system components implies, however, undesirable effects, such as non-determinism in the time domain and the value domain, which must be dealt with appropriately, in order to profit fully from redundancies.

The use of a time-triggered architecture significantly simplifies the synchronization effort required to overcome the non-determinism in the time domain. Section 4 gives an overview of the differences between the event-triggered and the time-triggered paradigms and describes the main properties of a time-triggered operating system and a time-triggered communication system (using the example of FlexRay).

In order to deal with non-determinism in the value domain, agreement protocols are required. In section 5, the relevant approaches for implementing these, as described in the literature, are presented and compared with respect to criteria derived from the requirements of safety-relevant electronic systems. The results of this comparison lead to the conclusion that the Signed Messages protocol is most appropriate in the scope of this thesis.

Based on the approaches analyzed, a concept for an Agreement Protocol service is suggested in section 6, taking into account the requirements for safety-relevant electronics and other constraints resulting from its integration in the EASIS software architecture. As well as a

description of the relevant algorithms, this section also includes a UML design which provides an overview of the different software units constituting the Agreement Protocol service and their dependencies.

In order to validate the suggested concept, a virtual prototype is developed in Matlab/Simulink, based on the time-triggered paradigm. Section 7 gives an overview of the tools that have been used and the steps required for the implementation and describes in detail the realization of the concepts presented in section 6 within the virtual prototype.

This prototype represents an instrument with which the Agreement Protocol service concept can be evaluated, since it allows for the simulation of different execution states. Therefore, several evaluation cases have been designed in order to assess the behavior of the Agreement Protocol service with respect to the relevant aspects, such as functionality, fault-tolerance and scalability. The evaluation results are presented in section 8.

Finally, in section 9, a summary of the work and an outlook for further steps within the EASIS project conclude the thesis.

2 Automotive Safety-Relevant Electronics

Nowadays a wide range of functions in vehicles are being realized as electronic systems, resulting in a strong and steady growth in the amount of electronics in a vehicle. With the introduction of ABS-Systems in 1978, developers in the automotive industry came to recognize the vast potential of using electronic systems to solve safety-related problems, which represented a major turning point primarily in the enhancement of automotive safety, but also in the evolution of automotive electronics (subsection 2.1).

The development of safety-relevant electronic systems differs from the development of other electronic systems in its rigorous requirements for fault-tolerance. Besides this aspect, subsection 2.2 also gives an overview about the technical requirements of safety-relevant systems.

In order to regulate the development process for automotive electronic systems, an appropriate model is required. Among the different development process models in use in the industry, the V-Model prevails in the automotive sector. This model must be expanded by safety-specific development steps and methods in order to be adapted to the development of safety-relevant systems (subsection 2.3).

The increasing amount and complexity of embedded software in electronic systems is one of the reasons for the standardization of the software architecture. Subsection 2.4 includes a detailed discussion about the motivation behind the standardization and presents the EASIS software architecture as a standard software architecture for safety-related application.

2.1 Electronic Systems in the Automotive Industry

Electronic systems currently represent the main field of innovation in the automotive industry. They are not only used to replace conventional mechanical systems but also to integrate new functions in vehicles. As a consequence, the share of electronics in modern cars is growing rapidly. Before discussing these aspects in this subsection, the general configuration of an electronic system will be described.

2.1.1 Components of an Automotive Electronic System

A typical automotive electronic system consists of: sensors, actuators, power supply, a communication system, micro-controllers and embedded software.

Sensors

Sensors are technological devices for measuring (sensing) physical dimensions, such as velocity, engine speed, temperature, pressure, acceleration, distance, angle, etc. They play an important role in electronic systems by providing the input data or signal. The signal provided is very often amplified before being processed by an ECU.

Micro-controllers

A micro-controller is a computer-on-a-chip, optimized to control electronic devices. Several components, such as CPU, ROM, RAM and I/O interfaces, are integrated on this chip.

Modern micro-controllers may also contain special blocks, such as clocks, Flash/EEPROM memory, communication interfaces, etc. The CPU can process data or signal input from sensors in real-time. The program software is stored in the microcontroller or other chips, typically in EPROMs and can be updated for re-programming. The output is typically sent to one or more actuators.

Actuators

Actuators are placed on the output side of an ECU. They convert received input signals (mainly electrical signals) into motion. Some examples of actuators are electric motors, relays, hydraulic pistons, etc.

Power Supply

The goal of the power supply section of an electronic system is to distribute power safely and effectively to each section of the system. The battery and the alternator are the main power sources in an automobile.

Communication Systems

Several bus systems and communication protocols have been developed within the automotive industry. They have different characteristics and application areas. Table 2-1 gives an overview about the state-of-the art in bus systems.

Bus System	**Description**
CAN	CAN (**Controller Area Network**) was developed by the Robert Bosch GmbH at the beginning of the eighties. It has been internationally standardized (ISO 11898) since 1994. CAN was specifically developed for the serial data exchange between electronic control units in automobiles.
LIN	LIN (**Local Interconnect Network**) is a serial communication system that is used for distributed electronic systems, usually sensors, in automobiles. Typically, a number of physically separated local LIN networks are connected with a CAN network.
MOST	MOST (**Media Oriented Systems Transport**) is a versatile, high-performance multimedia network technology based on synchronous data communication. It is ideal for multimedia applications in the automotive sector (audio, video, navigation and communication).
FlexRay	**FlexRay** is a manufacturer spanning bus system for high-speed applications. FlexRay was developed together by DaimlerChrysler and BMW in cooperation with semiconductor manufacturers. Features: • Synchronous and asynchronous data communication (scalable) • High data rate (5Mbit/sec); gross data rate approx. 10Mbit/sec • Fault-tolerant and time-triggered services are implemented in the hardware.

Table 2-1 Bus Systems [Vect05]

Embedded Software

Another name for embedded software in hardware devices is Firmware. Software is stored for example in ROM/Flash memory IC chips. Firmware is usually developed and tested more carefully than software for personal computers.

2.1.2 Functions and Complexity

A large number of electronic systems in the vehicle are used to replace or assist conventional mechanical or hydraulic systems. The power train domain as a whole (engine, clutch and gearbox) in most contemporary upper class vehicles (e.g. Mercedes-Benz S-class) and commercial vehicles is realized as a mechatronic system. For instance the mechanical coupling between the accelerator pedal and the throttle valve in the engine has been removed and replaced by the EGas-System [Hede01]. Besides replacing or assisting conventional mechanical systems, electronic systems are also being developed to add new functions and features as a response to the growing demand for safety and comfort.

Consequently, modern automobiles contain many complex electronic systems which incorporate a large number of electronic control units (ECUs) communicating through various layers of networks. These ECUs are becoming more numerous, complex, and interconnected. In every new vehicle generation, they handle an additional set of functions. In 2005, the number of ECUs in high end luxury cars (e.g. Maybach) reached more than 70 ECUs (see Figure 2-1). Accordingly, the percentage of an automobile's value attributable to the electronic content is also growing and is projected to reach 30% by 2008. [Hell04][Hell05].

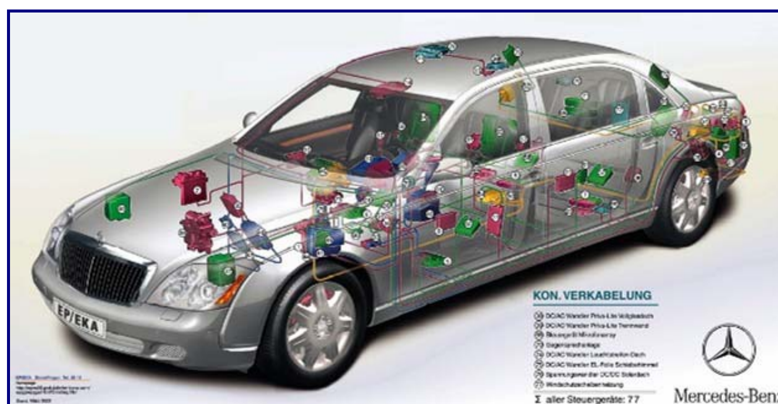

Figure 2-1 77 ECUs in the Maybach (luxury class) [Easi05a]

Besides hardware, electronic systems contain a large portion of embedded software, which is predicted to triple by the end of the decade. It is suggested that by 2010, more than 40% of the total value of automotive electronics will be in the software. Behind these increases is the reality that most ECUs will contain a huge amount of embedded software. Almost all the future technologies expected to appear in vehicles, such as drive-by-wire technologies and electrically operated valves, involve millions of lines of embedded software [Hell04]. There are two forces behind this evolution, as stated in [Voge03]:

- Software makes electronic systems cheaper
- New or improved functionality can be only implemented with the help of software

2.2 Safety-Relevant Automotive Electronic Systems

Due to the ever increasing volume of road traffic and high rate of road accidents, automotive safety has become a field of intensive research, offering a high potential for innovation. This led to many improvements in this field, mainly realized in electronics. In addition to conventional safety features, such as the ABS system or the ESP system, modern and future safety features include steer-by-wire, brake-by-wire and drive-by-wire (collectively known as “X-by-wire”), automatic lane-following, drowsy driver detection, intelligent cruise control and airbag systems that can adjust deployment based upon passenger weight and the specific nature of an accident.

2.2.1 Requirements of Fault-Tolerance

Most contemporary safety-relevant systems have a conventional mechanical backup or a safe operation modus (e.g. basic braking functionality for the ABS-System), so that redundancies are solely used for fault-detection purposes. This first step in realizing fault-tolerance only guarantees a *fail-safe* strategy, meaning that the system immediately switches to a safe state upon fault detection. The existing backup is mostly sufficient to reach this state.

To build safety-relevant systems without conventional mechanical backup, such as X-by-Wire systems, a fail-safe strategy is no longer sufficient. Therefore further fault-tolerance measures have to be taken. In fact, the system must continue operating and delivering its functionality despite faults. This extended behavior is called *fail-operational* [Hede01]. These aspects of fault-tolerance are discussed in more details in section 3.

2.2.2 Technical Requirements for Safety-Relevant Electronic Systems

In addition to fault-tolerance, some technical requirements, as described in the following, have to be fulfilled in order to build a safety-relevant system.

Low latency

The latency for a piece of information is measured by the time interval between generation and the completed use of this information. Its value can amount to a number of milliseconds and is always based on the application requirements. Constant and known latencies are a prerequisite for building safety-relevant systems [Hede01].

Predictability

The failure to meet timing constraints in a safety-relevant system can have fatal consequences. Therefore, these systems are said to be *hard real-time systems*. For such systems, predictability is most important and can be realized by means of *pre-run-time scheduling,* implying that the flow of processes and messages during operation is accurately predefined and does not depend on run-time events [XuPa93].

Fault Detection

Fault detection is particularly important in order to discover local faults or determine the components of a distributed system that are faulty and can no longer be relied on [Hede01].

Replica-Determinism

Fault-tolerance in real-time systems can be realized only by the means of active redundancy, meaning replicas of the same component operating simultaneously and providing the same functionality in a deterministic way. Therefore synchronization mechanisms are required to ensure that all replicas deliver their functionality within a specified time interval [Hede01].

Composability

Composability is a constructive approach to develop and implement complex systems. It consists in building up the system from well defined and validated subsystems or components. This is made possible by an unambiguous definition of subsystem interfaces and well defined responsibilities of the participants (e.g. OEM and suppliers), which enables system integration without side effects [Hede01].

Testability

Both testing of the system as a whole and the independent testing of single subsystems should be possible. The integration of a component in the whole system should not cause the necessity for retesting the local functionality of this component, so that only the whole functionality has to be tested upon system integration [Hede01].

Maintenance and diagnosis

Maintainability and the possibility for diagnosis are basic requirements for any ECU in the automobile and thus for any automotive electronic system.

2.3 Development Processes for Safety-Relevant Electronic Systems

In order to regulate the development process for automotive electronic systems, a systematic and standardized approach is required, called development process model. Among the different development process models in use in the industry, the V-Model prevails in the automotive sector.

The development of safety-relevant systems introduces specific safety-related requirements and constraints that have to be taken into account. As a consequence, the V-Model has to be adapted and expanded to cope with these new requirements.

Furthermore, approaches, such as model-based development, will enable a validation of system functionality in the early development stages, so that efforts, due to the late discovery of specification errors, can be saved.

2.3.1 The V-Model

The V-Model, first conceived in 1991 for German federal administration and defense projects, is a symmetrical model that has become standard in software development. It consists of 2 wings as depicted in Figure 2-2. The left wing describes the implementation path, starting out from high-level requirements and becoming more detailed at every step up to the actual implementation, while the right side represents the validation path, in which each validation phase corresponds to an implementation phase on the right wing.

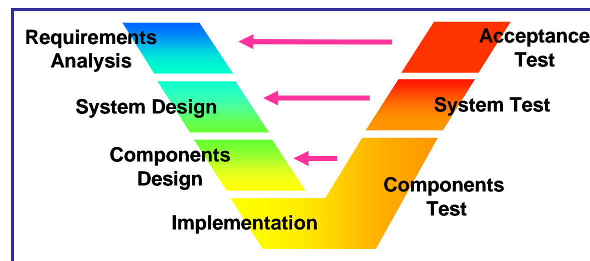

Figure 2-2 V-Model

The disadvantage of this model is that all system components have to be implemented first, before any of the subsystems can be tested, which corresponds to a *bottom-up* approach. As a consequence, no validation can be undertaken in early development stages. Moreover the future user cannot really get a feel for how the system will look, until the development is almost complete [Hede01].

Nevertheless, this process model is well established in the industry, as it seems to be most adequate, providing a well defined approach for the development of systems of different complexities. Furthermore, this model enforces a full analysis and specification of the whole system before the actual development begins, which is a necessary prerequisite for large projects involving different parties.

2.3.2 Safety Relevant Development Steps

The use of the V-Model ensures that the end product complies with the functional requirements defined at the beginning of the development process. In order to deal with safety-related requirements the V-Model has to be further expanded by safety-specific elements. [Benz04] describes an approach, where safety-specific activities are merged in the phases of the V-Model, giving rise to a new form of this model called the *double V-Model* (see Figure 2-3). This approach is briefly presented here.

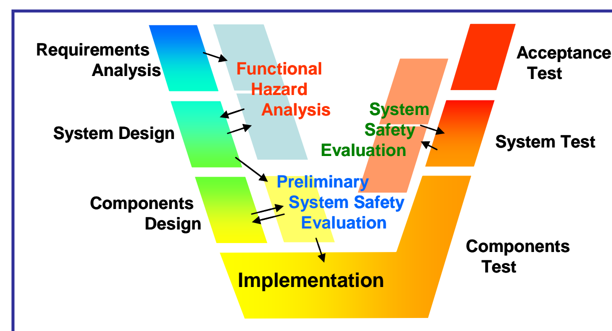

Figure 2-3 Double V-Model

During the functional requirement analysis and system design, a functional hazard analysis has to be carried out, consisting of identifying and evaluating the possible impact of the failure of each system function and sub-function. This evaluation should comply with standards and constraints specific to the automotive industry, which can be, for example, guaranteed by the use of the *CARTRONIC Safety Analysis*. This extensive hazard evaluation assigns a reliability level for each system function and sub-function.

The requirements delivered by the functional hazard analysis must be taken into consideration during the elaboration of a system concept and the design of the system architecture in the scope of a preliminary system safety evaluation. Methods such as *FMEA*, *Dependability diagrams* and *Markov analysis*, enable a quantitative analysis and evaluation of alternative system concepts and architectures as regards their compliance within the defined safety requirements.

The system validation in the V-Model should also be combined with safety-specific measures within the scope of a system safety evaluation. The evaluation methods used here are the same as in the previously introduced steps, the main difference being that they are now applied on a real and physically implemented system, rather than system concepts and system architectures. This evaluation aims to prove that the implemented system actually meets the safety requirements.

These additional safety-related analysis and validation steps are integrated in the V-Model by adding them onto both of its wings. This results in the creation of new safety-specific paths on both wings of the V-Model as shown in Figure 2-3.

2.3.3 Model-Based Design using the Example of Simulink

Safety-relevant electronic systems are complex products, as they often combine distributed hardware and embedded software that have to be developed harmoniously and in shortest time in order to be able to keep up with the market. Therefore, they require a systematic design approach such as the model-based design presented here.

Model-based design, also called *electronic system-level* (ESL) design, consists of creating an executable software model on system level, called *virtual prototype,* where all functional and non-functional requirements for the system are expressed. This approach, based on advanced modeling and simulation techniques such as UML and Simulink, enables an accurate prediction of the system behavior long before it physically exists and hence offers the possibility to quantitatively evaluate different architectural options and tradeoffs with a relatively small overhead. [Hell04] presents this approach and outlines its benefits, which are briefly described here.

The model-based design solves a major problem in the development of electronic systems involving hardware and software components. The development of such systems requires the close cooperation of hardware and software developer teams. This induces in most cases a considerable time overhead, since these teams are very often geographically dispersed. The model-based approach provides an effective and efficient means of communication (i.e. the model). In fact, the system specification is formally translated into a well defined model, so that any changes to the system requirements are reflected in the model after being approved by all parties. This model constitutes therefore a “golden reference” for both hardware and software developers.

Moreover, the virtual prototype (i.e. the model) provides embedded software developers with a platform, which conforms to reality, where they can easily debug and validate their implementation without waiting for a hardware prototype, as opposed to the bench testing

approach. This makes the simultaneous development of embedded software and hardware components possible, which reduces the overall development time significantly.

Different development tool chains currently exist for the model based design, such as Simulink, which is widely used, especially for the development of automotive electronic systems. Simulink is a platform for multi-domain simulation and model-based design of dynamic systems. It provides an interactive graphical environment and a customizable set of block libraries that allow for accurate design, simulation, implementation, test control, signal processing, communications, and other time-varying systems (see Figure 2-4) [Math05a].

This development environment offers the possibility of building a model consisting of blocks, where each block delivers a specific function and has a well defined interface to the other blocks. Besides the standard Simulink blocks (e.g. MUX, type conversion, etc.) user-defined blocks can be implemented to realize more specific functions. The user may choose among ADA, C and C++ for the implementation of user-defined blocks in the form of S-Functions.

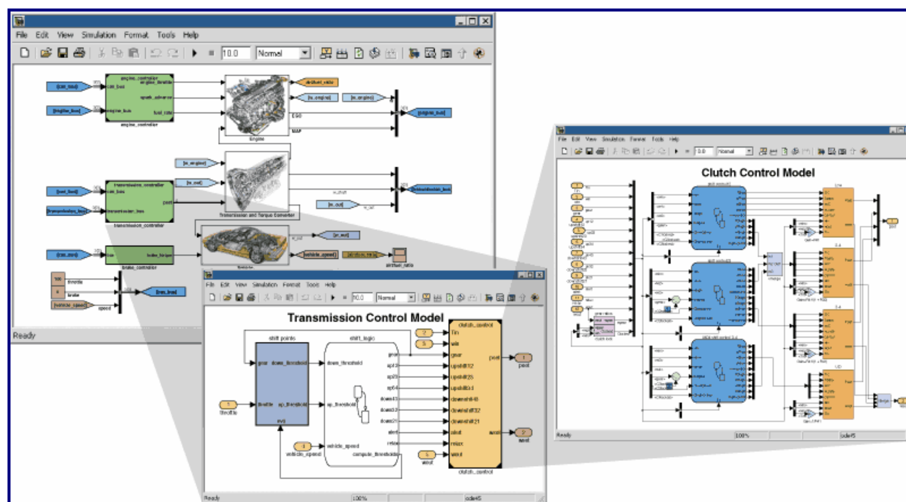

Figure 2-4 Simulink by "The MathWorks" [Math05a]

Furthermore, some suppliers and software companies in the automotive industry are working to enhance Simulink by providing customized block libraries which can be added easily to the default Simulink block libraries. Such efforts make it possible to integrate the latest standards in Simulink modeling and simulation environment and therefore to keep up to date with technological progress. An example of such a company is DECOMSYS which offers a Simulink block-set based tool-chain for the design of FlexRay based applications. Some products of this tool-chain are used together with the standard Simulink library in the scope of this work. They are briefly described here, according to the corresponding product sheets of DECOMSYS [Deco05]:

DECOMSYS::SIMSYSTEM<FlexRay>

DECOMSYS::SIMSYSTEM is a block set for use with Simulink which enables the modeling of hard- and software architectures, distributed in FlexRay systems. Furthermore, it facilitates the simulation of temporal behavior of task activations in time-triggered operating systems, such as OSEKtime/OS.

DECOMSYS::SIMCOM<FlexRay>

DECOMSYS::SIMCOM is a FlexRay communication system block set for use with Matlab/Simulink. It contains blocks to transfer Simulink signals over the FlexRay bus. A high-performance protocol engine simulates the communication system in the background and allows the developer to explore the effects of communication in the distributed application.

DECOMSYS::DESIGNER

DECOMSYS::DESIGNER is not a block set but a tool that can be used in combination with the two previous ones. It eases the development of distributed real-time systems based on automotive networks, e.g. FlexRay, by accompanying the user through the entire development process. A graphical user interface allows the user to design the architectural models of the system. Based on these models DECOMSYS::DESIGNER builds the communication schedule and configures the communication system.

Models built in Simulink can be configured and made ready for code generation. Using Real-Time Workshop, code can be generated from the model for rapid prototyping or deployment [Math05a].

2.4 A Standard Software Architecture for Integrated Safety Applications

Electronic systems contain, besides hardware, a large portion of embedded software which is increasing at a fast pace. In the absence of a standard approach, the complexity of embedded software is increasing and becoming harder to manage. These challenges can only be overcome through the introduction of a standardized embedded software architecture, which not only aims to master the complexity, but will also offer major benefits.

Recognizing the need for standardization and its benefits, OEMs and suppliers have decided to combine their efforts in the scope of several projects aiming at the establishment of an industry-wide standard software architecture. The EASIS software architecture, based on a more general standard software architecture, is designed to support particularly the development of integrated safety applications.

2.4.1 The Motivation for Standardization

The development of an electronic system often involves several parties (OEMs, suppliers, etc.), providing hardware and software solutions to build the system. In the absence of a standard approach, this has led over the years to a highly complex and heterogeneous electronic landscape in vehicles, characterized by standalone and proprietary solutions. The software implemented depends to a large extent on the specific interfaces and architecture of the hardware platform used, since an appropriate abstraction level, i.e. a clear separation between application-related and hardware-related software, is missing. As a consequence, the integration of an implemented function in a different hardware platform requires significant efforts and might be very expensive. Furthermore, it is not possible to exchange a particular software implementation with another software solution, as the new solution is not necessarily

compatible with the hardware platform. These drawbacks reduce the design flexibility of electronic systems significantly, leaving little room for changes and improvements [Voge03].

Recognizing these challenges and the resulting limitations, leading OEMs and suppliers decided to work together on creating a standard embedded software-architecture for embedded software. The result is a framework with all basic services and interfaces (e.g. OS, COM, NM, I/O, etc.) provided in an abstract form, i.e. independently from the underlying hardware, to the application level. Within this standard framework, developers can make use of pre-implemented basic services, which saves effort and costs.

Moreover, the use of a standard API to access basic services results in hardware independent implementation, so that the produced software can run on different hardware platforms. This enables, in particular, a component-based implementation of the application, where single software components can be easily ported to other platforms. As a consequence, the application is not bound to a specific hardware-topology (i.e. a constellation of specific ECUs), meaning several options can be evaluated. This is a great benefit for the development of distributed applications, since they can be far more easily partitioned over several ECUs.

In the following, two industry wide projects, EAST-EEA and EASIS, are presented. These, although having different names, are not different alternatives. They rather follow the same standardization goals and complement each other.

2.4.2 EAST-EEA

The EAST project (Electronics Architecture and Software Technologies) was launched in mid 2001 with the aim of specifying a standardized open software architecture framework for electronic systems. This architecture is middleware based, meaning that hardware and communication services are abstracted in a common interface which can be uniformly accessed by the application layer through a standard API. As depicted in Figure 2-5, the basic functions embedded in the EAST architecture include communication drivers, I/O drivers, memory drivers, I/O services and communication services, while OSEK is suggested as operating system. OSEK is a real-time operating system developed by a consortium from the automotive industry (OSEK/VDX) and has therefore become standard for automotive applications. All standards defined by OSEK/VDX are described in detail in [Osek05].

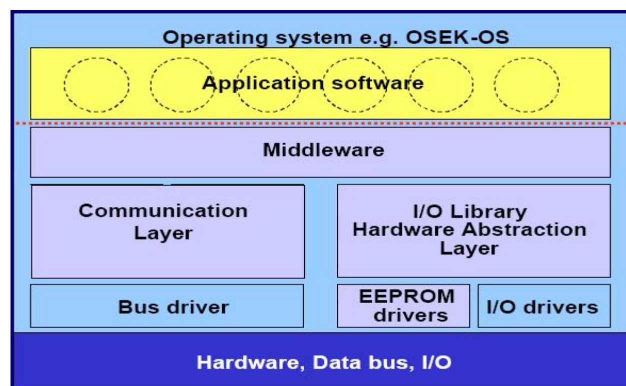

Figure 2-5 EAST architecture [IrJu04]

The implementation of application components on top of the EAST-Middleware can be achieved separately from a target hardware platform. This not only enables the allocation of components from different applications on a single ECU, but also allows for the partitioning of functions over several ECUs.

Even though the EAST-EEA project ended in 2004, the work continues. The results of EAST-EEA are being used as the basis for the EU Sixth Framework Program project EASIS, presented in the next subsection.

2.4.3 EASIS

The EASIS project, launched in 2004, focuses on safety-relevant aspects in the scope of an extended standard software architecture (see Figure 2-6). This in-vehicle architecture should enable the integration of safety systems and their combination with upcoming enhanced telematic services into a complete network of so-called Integrated Safety Systems. The EASIS architecture integrates, besides the basic services, additional standard services, such as the Gateway Services and the ISS Dependability Services, where the Agreement Protocol will be embedded. These software services aim to enhance the safety, reliability, availability and security of new safety functions on the one hand, and provide basic concepts for software gateway features (e.g. firewalls) on the other.

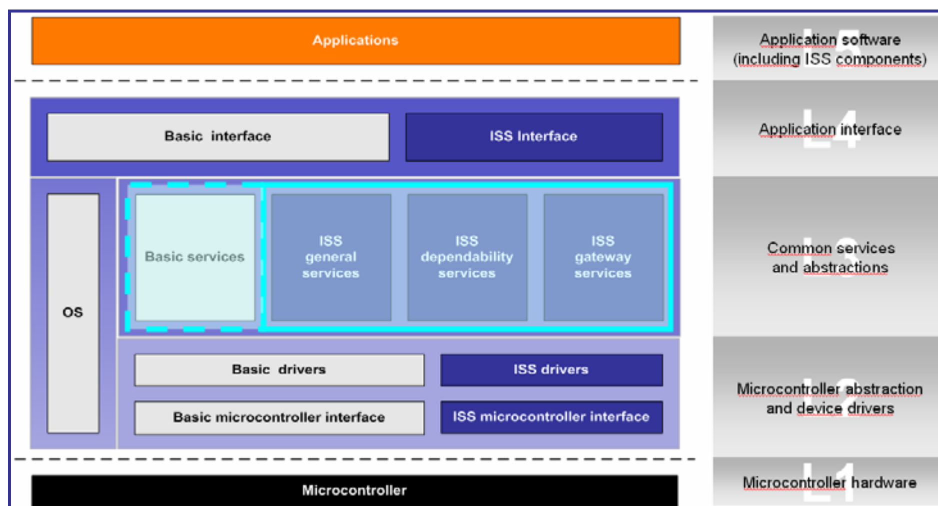

Figure 2-6 EASIS software architecture

Following the same principle of standardization, all the services in EASIS are offered to the application layer through an application interface. Based on this standard interface, new safety functions can be developed in a shorter time, for lower system costs and with an improved system quality [Easi05b].

3 Aspects of Fault-Tolerance in Safety-Relevant Electronic Systems

Although developers put significant efforts into the design and testing of electronic systems to avoid the occurrence of faults, it is very often hard to ensure and prove that a system will actually never fail during operation. Therefore, fault-tolerance mechanisms, such as the use of structural redundancies combined with a fault-masking mechanism, are almost inevitable for safety-relevant electronic systems with high requirements for dependability (subsection 3.1).

Depending on the level of fault-tolerance to be achieved, different architectures exhibiting structural redundancies are possible. The relevant architectures are briefly presented in subsection 3.2.

3.1 Basics of Fault-Tolerance

Before addressing the topic of fault-tolerance, a few relevant terms are defined, mainly in accordance with [BuWe01].

3.1.1 Failure, Error and Fault

A system *failure* is observed, when the behavior of the system deviates from its specification. There are three basic kinds of failures. *Value failures* affect the value domain and consist of an incorrect result being delivered or a faulty parameter being passed to some function, resulting in an unexpected behavior. *Timing failures* consist in the late, early or omitted service delivery. Finally, *commission failures* are observed when a service is unexpectedly rendered. These failures might occur separately or in combination, resulting in the following system behaviors:

- ***Fail-late:*** deadlines being missed
- ***Fail-silent*:** correct behavior of a service and all subsequent services, before failure; upon failure, these services are not rendered
- ***Fail-stop*:** like fail silent, but service requestors (i.e. other systems) can detect that the system will no longer respond
- ***Fail-uncontrolled:*** the failure results in a byzantine (i.e. arbitrary) behavior in both value and time domain

While referring to the external system behavior, a system failure originates internally, due to system internal *errors*. These arise, when the system reaches an internal state that was not anticipated in its design. The technical cause (physical or algorithmic) for an error is termed a *fault*.

Faults can be classified in three categories:

- ***Transient faults:*** are induced by transient external influences such as heat
- ***Permanent faults:*** e.g. coding errors and hardware damage

- ***Intermittent faults:*** occur sometimes and are unpredictable

A *fault assumption* defines, according to [Easi04a], the set of faults to be detected or tolerated in a system. It therefore specifies the types of fault origins as well as their location and the errors caused by them. In the scope of the EASIS project, the following fault assumption is made, as described in [Easi04a],

- ***Local hardware faults*: ** are due to aging and disturbances. The faults can either be temporary or permanent. In both cases the resulting errors do need to be constant. Instead they may cause various sequences of erroneous states. Moreover, the faults may propagate to neighboring hardware or software components.
- ***Global hardware faults*: ** cause temporary blackouts of a number of components. These faults may stretch over several nodes and communication links. They might be caused by electromagnetic interference.
- ***Software design faults*: ** are mainly considered with respect to the development process. Design diversity is excluded from EASIS, to avoid a costly widening of the project. Moreover, design diversity does not appear to be typical for automotive software.

In order to deal with faults, two approaches may be adopted. *Fault prevention* is the first of these approaches and focuses on the development process to achieve faultlessness. It includes fault avoidance (e.g. most reliable techniques, tools, components, etc.) and fault removal (e.g. V&V, testing, etc.).

Dijkstra said, “Testing can only show the presence of errors, never their absence”. Therefore, *fault-tolerance* is required to deal with the remaining potential of faults.

3.1.2 Fault-Tolerance, Reliability and Safety

The increase of automation in technical systems is combined with a progressive shift of responsibilities from human to machine, resulting in far higher requirements for the reliability and safety of these systems. As defined in [BuWe01], reliability is the measure of success with which the system conforms to the specification of its behavior. It refers therefore to the correctness of system operation. Safety designates, according to [Echt90], the absence of danger to the environment during system operation or upon system failure and relates therefore to the nature of the system (e.g. involving human life or not). While achieving high accuracy and high execution speeds and enabling an autonomous operation, automated systems are neither intrinsically reliable nor safe. On the contrary, since their complexity is continuously increasing, they are far more susceptible to faults and thus failures. If no special measures are taken, these systems cannot be relied on in case of faults and could therefore represent a potential danger for the user and environment.

The idea of developing systems in such a way, that they are still able to provide their functionality even upon the occurrence of faults, has existed in a variety of technical (e.g. the spare wheel in a car) and biological systems (e.g. two eyes) for years [Echt90]. This idea is reflected in the principle of fault-tolerance, which is defined in [BuWe01] as the ability of a

system to handle internal errors and faults in such a way that its failure is avoided or at least less severe with regard to its consequences. Fault-tolerance mechanisms are therefore continuously gaining in importance, since they represent a great instrument to ensure reliability and safety despite faults. In the scope of safety-critical systems, such as X-by-Wire systems, it is not possible to consider reliability and safety separately, since the latter is the immediate result of the former. Consequently, by applying fault-tolerance mechanisms to improve system reliability, safety will be also guaranteed.

Depending on the system behavior in the presence of faults, at least three levels of fault-tolerance can be distinguished, as described in [BuWe01]:

- ***Fail operational (full fault tolerance):*** the system keeps on operating upon the occurrence of faults, without significant loss of functionality or performance
- ***Fail degraded (graceful degradation):*** the system continues operation in the presence of faults. However, a partial degradation in functionality or performance is observed.
- ***Fail safe:*** the system ceases to operate upon occurrence of faults in such a way, that its environment is left in a safe state, in which the potential of damage is minimized.

3.1.3 Structural Redundancies and Non-Determinism

One way of realizing fault-tolerance in technical systems is the use of *structural redundancies*. These are defined in [Echt90] as the expansion of the system by additional components (software or hardware), which are not required for the actual system function. Furthermore, they are solely replicas of existing system components, with respect to their function or even to their implementation and contribute therefore, towards fault-tolerance only.

Within an electronic system including structural redundancies, different sources of undesirable non-determinism can be identified, as discussed in [EcBe02]. One form of non-determinism affects the time-domain and arises in the presence of a high latency jitter in the communication system, so that neither the arrival time of redundant messages nor their sequence in time can be predicted accurately. Therefore, synchronization is required (e.g. in the communication system) to guarantee a simultaneous processing of redundant messages. The use of a time-triggered architecture, consisting of a time-triggered operating system and a time-triggered communication system, simplifies the synchronization effort significantly. The time-triggered is discussed in detail in section 4.

While measuring the same dimension using redundant sensors, the measured values, which theoretically should be identical, may exhibit slight deviations due to the physical nature of the sensors and are therefore non-deterministic. The use of these values leads, of course, to the computation of non-deterministic results, which is not acceptable for safety-related applications. For example, a simple majority voter would not be able to distinguish between non-deterministic redundant results and faulty ones and would consequently consider all values as being faulty, which makes the redundancy in this case useless. In order to deal with this form of non-determinism, called non-determinism in the value domain, fault masking, as described in the next subsection, is required.

3.1.4 Fault Masking for Structural Redundancy

In order to make optimal use of redundancies, different strategies can be followed. The *backup strategy* says that the results of duplicating components (i.e. redundancies) are ignored until the primary component fails. According to the *voting strategy,* the results of all redundant components are compared and the majority result wins [BuWe01]. While the voting strategy defined here is limited to the majority voting to decide the final result, *fault masking*, a more general approach discussed in detail in [Echt90], does not imply the use of any particular decision strategy.

Fault masking assumes the presence of a *k-out-of-n system* (see Figure 3-1). This consists of *n* redundant components, at least *k* of which must be non-faulty to ensure fault-tolerance (in this case, fault masking). The input data is replicated and distributed to the redundant components for processing. Each component delivers its result to the *masker*, where some kind of voting, termed *masking decision*, is carried out in order to decide on the final output of the system. The masker therefore represents the basic element of a fault-masking system, enabling fault-handling and fault diagnostic [Echt90].

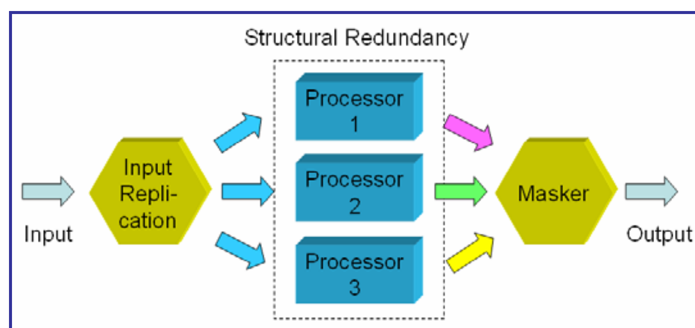

Figure 3-1 A 2-out-of-3 System [Echt90]

Given the high requirements for reliability of some applications, the masker, which constitutes a single point of failure, might be also replicated. Since additional communication is required to reach a consistent decision among maskers, the overall messages overhead is increased. The message exchange in the scope of fault-masking is called *masking protocol*.

3.1.4.1 Masking Decisions

A masking decision is carried out by the masker in order to decide on a final output, in the presence of redundant results. While several kinds of masking decisions can be implemented, the decision process is always based on *relative tests*, which consist of comparing the redundant results in pairs. This enables not only the masking out of faults, but also the diagnosis of their origin.

Masking decisions are primarily subdivided into two categories, depending on whether determinism among the redundant information is given (see Table 3-1). The different decision strategies are described here using the example of an n-component system, as in [Echt90]. It is assumed for reasons of simplicity that $n = 2m + 1$.

Processes exhibiting replica determinism	Processes exhibiting replica non-determinism
Majority decision	Median decision
Mate decision	Interval decision
Quasi-majority decision	Sphere decision
Unanimity decision	

Table 3-1 Masking decisions [Echt90]

Majority decision

The redundant results are classified in equivalence classes, so that all results in an equivalence class are equal. If a class contains the majority (i.e. at least $m + 1$) of the results, it is considered faultless and its result is delivered. Therefore, no more than k faults are tolerated.

Mate Decision

The mate decision is based on the assumption that two faulty results are never equal. Therefore, a pair of results is considered faultless, if the two results are equal. One of these results is then randomly chosen and delivered.

Quasi-Majority decision

The quasi-majority decision represents an extension of the majority decision in the case where the majority of the results are faulty. In this case, the equivalent class with the most results is considered faultless and its result is delivered. This implies that more than m faults can be tolerated by the quasi-majority decision.

Unanimity Decision

Unanimity decisions do not guarantee fault-tolerance, but solely fault detection. A value is only delivered if all the redundant results are equal. The unanimity decision is often implemented for 2-out-of-2 systems, such as duplex systems.

The presented masking decisions all assume determinism of the results in the value domain. Some applications, however, exhibit a certain non-determinism which cannot be avoided and has therefore to be taken into account, as in the following masking decisions.

Median Decision

The median decision can be used, when the set of results can be ordered and mapped to a linear interval. The decision then consists of ignoring the lowest m values and the highest m values in the interval. Finally, the remaining value is considered faultless and is therefore delivered. At the example of a 2-out-of-3 system delivering the values 19, 17 and 54, 17 and 54 are ignored, while 19 is delivered.

Interval decision

The interval decision, which supports multidimensional values, defines for each dimension i the maximal width δ_i of the interval, to which correct values of the i-th dimension might belong. The interval bounds are left variable. If the components of at least k values lie in

intervals complying with the defined properties, then a value is randomly chosen and delivered. The interval decision is well suited for sensor measurements, where correctly measured values can not be expected to be identical.

Sphere decision

The sphere decision is analogous to the interval decision with the difference that it defines for all dimensions a single maximal width for the intervals of correct values.

3.1.4.2 Masking Protocols

According to [Echt90], two classes of masking protocols can be identified.

Ignoring protocols should allow the maskers to ignore faulty results, when determinism is given. The remaining results are then all equal and correct.

In the presence of non-determinism, *agreement protocols* should enable the maskers to reach agreement with regard to the choice of a correct result. Agreement protocols are discussed in detail in section 5.

3.2 Fault-Tolerant Architectures

Given the high requirements with regard to the reliability of safety-critical systems, such as the X-by-Wire systems, several parts of the system might be subject to replication, such as sensors, ECUs (i.e. processing units) and actuators. The replication of hardware, while improving reliability, contributes towards increasing costs. Therefore trade-offs are made depending on the system requirements. These are in turn reflected in different fault-tolerant architectures.

3.2.1 Duplex Systems

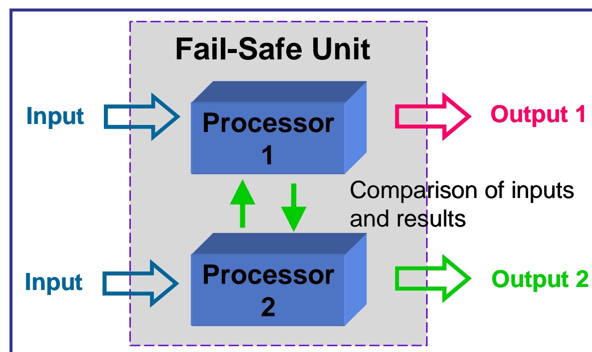

Figure 3-2 A duplex system

A duplex system consists of two processors, each computing a result, as depicted in Figure 3-2. If both results are equal, they are considered to be correct. Otherwise, both results are ignored. Consequently, duplex systems are not fail-operational, since a single faulty processor would cause the results to be non-consistent and would therefore lead to a safe deactivation of the system, in order to avoid the propagation of faulty results. Therefore, such systems are only fail-safe [EcBe02].

3.2.2 Fault-Masking Dual Duplex Systems

A dual duplex system (see Figure 3-3) consists of two redundant duplex systems (i.e. two fail-safe units) operating simultaneously and each providing a result, if they are non-faulty. If one of these units fails, the system can rely on the other to continue operation. Thus, both fail-safe units together form a fail-operational unit, which is able to tolerate a single faulty processor [EcBe02].

It is for instance possible to transfer both results over redundant communication paths to the target component, which only considers the first correctly received result and ignores the other.

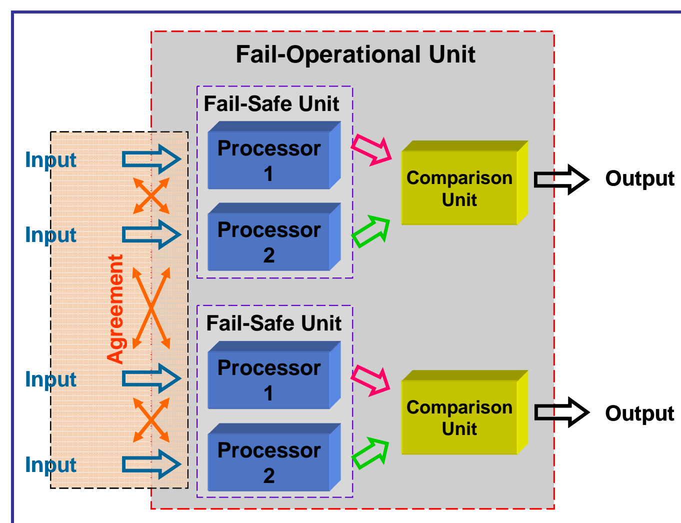

Figure 3-3 A dual duplex system [EcBe02]

3.2.3 Fault-Masking Triplex Systems

As shown in Figure 3-4, a triplex system consists of three redundant processors, each computing a result, from equal input data. In the presence of three results, a majority decision (i.e. 2 out of 3) can be made, therefore tolerating one faulty value and thus one faulty processor. Consequently, triplex systems are operable as fail-operational units [EcBe02].

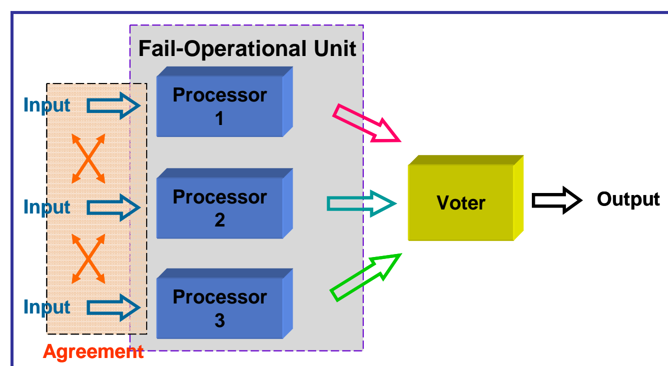

Figure 3-4 A triplex system [EcBe02]

4 Time-Triggered Architecture for Safety-Relevant Electronic Systems

Depending on system requirements, electronic systems can be implemented based on an event-triggered and/or a time-triggered architecture. The event-triggered paradigm is appropriate for systems with a dynamic behavior, while the time-triggered paradigm is more suited for systems with a deterministic, cyclic behavior. Considering their requirements for predictability and determinism, safety-relevant electronic systems very often adopt a time-triggered architecture (subsection 4.1).

The main building blocks of a time-triggered architecture are a time-triggered operating system and a time-triggered communication system. The time-triggered operating system provides, besides basic operation services, specific functions to ensure a time-triggered, synchronized execution of tasks in each system node according to a predefined execution schedule. The time-triggered communication system ensures the time-triggered, deterministic message exchange between distributed system components (subsection 4.2).

FlexRay, a time-triggered communication system, has been developed by an industry-wide consortium to comply with the requirements of modern electronic applications. In order to provide a high flexibility, FlexRay not only implements the static time-triggered approach, but is also operable as a dynamic event-triggered communication system. Moreover, it provides mechanisms to improve communication reliability and thus simplify the implementation of protocol-based dependability services (subsection 4.3).

4.1 Time-Triggered vs. Event-Triggered Systems

Real-time systems can be designed according to two different paradigms: the event-triggered and/or the time-triggered. These paradigms and their main differences are briefly presented here, as described in [Hede01].

An event-triggered real-time system responds to events originating in the system itself or its environment. These events, which generally can not be foreseen, induce interrupts in the system CPU, where the required processing is then carried out. To resolve conflicts, due to the simultaneous occurrence of two or more events, a priority mechanism is often required. Hence, the system exhibits a dynamic behavior. The event-triggered approach is simple and sufficient for *soft real-time systems*, where missed deadlines are undesirable but not fatal. Problems could arise, however, when events cannot be predicted and their frequency goes up rapidly, overloading the CPU. This can cause the loss of some events, or, in the worst case, the failure of the whole system. This is why, the event-triggered approach is considered to be non-deterministic.

This cannot happen in time-triggered systems, as these operate according to a pre-defined time table, called a schedule. A schedule specifies, which actions (e.g. the reading of sensor information, the starting of computations or the activation of certain actuators) are to be undertaken at which point in time. These actions are then cyclically triggered by the elapsing

of pre-defined time segments, without the need for a priority mechanism. As synchronization is time-based, a global clock is used to provide all system nodes with consistent time information to guarantee a synchronized, system-wide operation. Time-triggered systems therefore exhibit a static and deterministic behavior. By allocating enough execution time for each action, there will be no risk of system overload.

Both the event-triggered and the time-triggered approaches can be combined in a single system, depending on the requirements of system sub-functions. For instance, the sub-function ABS in an ABS ECU is time-triggered with an execution cycle of 20 milliseconds, while the communication with the engine ECU is realized by means of an event-triggered communication bus (CAN).

As discussed in subsection 2.2.2, safety-relevant electronic systems must exhibit a predictable and deterministic behavior. Moreover, in the presence of redundancies, replica-determinism has to be guaranteed. The time-triggered approach appears to be more appropriate to satisfy these requirements, since the behavior of each system node and the communication scheme in the system can be statically pre-specified. In order to implement this approach, two principle subsystems must be provided: a time-triggered operating system (e.g. OSEKTime) and a time-triggered communication system (e.g. FlexRay, TTCAN, etc.).

4.2 Time-Triggered Operating Systems

The operating system provides real-time applications with basic services, i.e. task management, system time and clock synchronization, local message handling, error detection mechanisms and - if necessary - interrupt handling. In the scope of a time-triggered system, the operating system must additionally exhibit a predictable and static behavior to guarantee the determinism and predictability of the whole system. As a consequence, dynamic task activation, for instance, is not possible. In the following, some of the most relevant services offered by a time-triggered operating system are briefly described, using the example of OSEKTime, as in [Ring01] and [HRK+00].

The task management in a time-triggered operating system includes the time-triggered activation of tasks and the monitoring of deadlines. The time table (i.e. the task schedule), which is locally stored, is cyclically executed in *dispatcher rounds*, whereby tasks are activated by a *dispatcher* in a strictly sequential order and are assigned their pre-defined execution time. If a task is still running at the activation time of a new task, the old task is pre-empted and remains so, until termination of the newly activated task, which is referred to as *stack-based scheduling*. During task execution, the execution time of each task is monitored in order to detect deadline violations and notify the application for error handling.

Furthermore, the time-triggered operating system is responsible for time synchronization with the system global time to make sure system nodes operate synchronously. The time synchronization is carried out at system start-up and after loosing the synchronization with the global time base. A periodical synchronization during normal system operation is also required to guarantee that local clocks do not drift apart.

4.3 Time-Triggered Communication Systems using the Example of FlexRay

Since many applications in a time-triggered system are distributed over more than one ECU, a communication system is required to enable a reliable and deterministic message transmission between distributed components. The communication system FlexRay has been developed by a consortium of 98 companies in the automotive industry, to comply with these requirements. Combining a synchronous and an asynchronous communication scheme, it provides not only the required determinism but also flexibility. Hence, it is considered as a substitute for event-triggered communication systems such as CAN, while achieving a much higher data rate (10Mbit/s). As a consequence, FlexRay is becoming standard in the automotive industry.

The FlexRay communication system exhibits the following main characteristics, as in [Seth04]:

- Support for redundant transmission channels
- High data rate, up to 2 x 10Mbit/s
- Error containment on the physical layer through an independent bus guardian
- Support for bus topology and/or active star topology
- A time-triggered behavior, based on system global time or synchronization with an external trigger
- Synchronous and asynchronous data transmission
- Deterministic data transmission, guaranteed message latency and message jitter

4.3.1 Architecture of a FlexRay Node

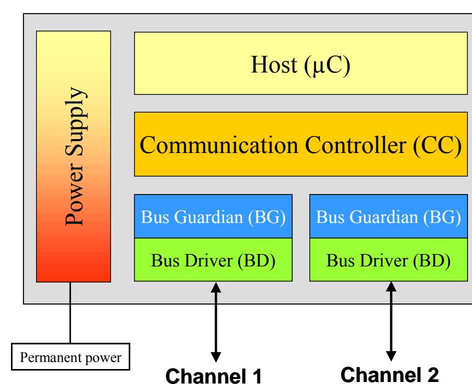

Figure 4-1 FlexRay node architecture

The general architecture of a node is depicted in Figure 4-1. The *communication controller*, where the FlexRay protocol is executed, represents the main component in this architecture. It can be connected to 2 different communication channels, each of which supports a data rate of 10 Mbit/s. These can be exploited either to double the transferred data volume or to enhance fault-tolerance by means of redundant communication paths.

An independent *bus guardian* contributes towards error containment by ensuring that no data collisions occur, even in the case of a faulty communication controller, and that no participant

monopolizes the communication system (referred to as a babbling idiot). In fact, by activating and deactivating the *bus driver*, according to a replicated communication schedule, the bus guardian can detect and cut off faulty accesses to the communication channel initiated by a faulty communication controller. As shown in Figure 4-1, each communication channel is protected by a bus guardian [HSB+02].

4.3.2 Network Topologies

FlexRay supports two kinds of interconnection topologies: a bus-based topology and a star-based topology (see Figure 4-2), which can be also combined, resulting in a variety of options. In both cases, FlexRay can operate not only as a single channel system such as CAN and LIN, but also as a dual channel system. In order to attain its high transmission speeds, the active star configuration is deployed.

Active stars consist of a node containing multiple bus drivers linked by a high speed back plane. They are used to build up communication paths out of point-to-point connections and hence contribute towards high data rates as well as enhanced fault containment in a communication system as a whole. Furthermore, when faults are detected on a communication branch, the active star disconnects this branch, thus stopping the propagation of faults to the rest of the system [HSB+02].

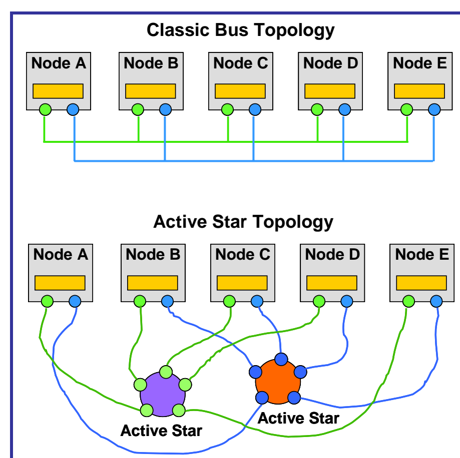

Figure 4-2 FlexRay topologies [HSB+02]

4.3.3 FlexRay Communication Protocol

The communication within FlexRay occurs in recurring *communication cycles*. The length of a communication cycle can either be fixed or variable. In the latter case, it depends on an external, cyclic trigger (e.g. the rotation of the crankshaft). This requires the knowledge of the minimal cycle length, which has to be greater than the maximal message propagation delay in the system. By fulfilling this condition, it is ensured that messages from different cycles do not collide.

In order to avoid collisions between messages sent from different system nodes on a shared communication channel, two different medium access mechanisms are implemented in FlexRay, providing support for a deterministic time-triggered, as well as a dynamic event-triggered, operation:

- TDMA: Time Division Multiple Access
- FTDMA: Flexible Time Division Multiple Access

These two mechanisms are combined in the communication cycle, where they are implemented in a *static segment* and a *dynamic segment* respectively, as shown in Figure 4-3.

The static segment of a communication cycle is subdivided into slots of equal length. A static schedule, computed offline during system design, assigns each FlexRay node one or more slots on each channel, where this node is allowed to transmit messages (TDMA). Since the sending time of each node is pre-specified, the data transmission within the static segment is deterministic, which is why this segment is more appropriate for cyclic and safety-critical messages requiring bounded latency and small latency jitter.

As opposed to the static segment, there is no static assignment of bandwidth within the dynamic segment. The sizes of time slots are also not static. Different nodes may compete for bandwidth using a priority-driven scheme. Moreover, the segment is divided into mini-slots corresponding to decreasing priority levels, so that each slot allows only one particular priority level. A node holding a frame with a certain priority level must wait for the corresponding mini-slot to transmit this frame. If transmission is initiated, the next mini-slot will be delayed until the transmission is done (FTDMA), so that low-priority mini-slots may not occur due to the time limitation imposed by the cyclic behavior. The bandwidth is then assigned according to the message priority. A high-priority message is guaranteed to be sent during the dynamic segment, while low priority messages are delayed and might not be sent if the segment length does not allow for it.

The dynamic segment is therefore suitable for ad-hoc data communication with varying bandwidth requirements, such as those resulting from an event-triggered behavior.

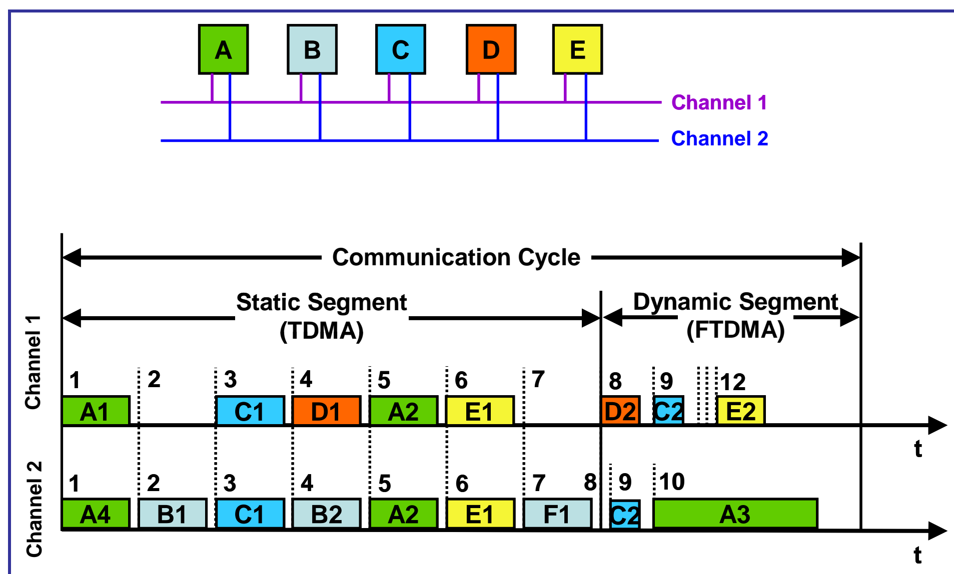

Figure 4-3 FlexRay communication cycle [HSB+02]

The configuration of the communication cycle (e.g. length of segments, length of slots, assignment of slots to nodes, priorities, etc.) is described in a schedule which is then cyclically executed by the communication controller. The schedule is additionally replicated

in the bus guardian, to enable the monitoring of the communication controller's accesses to the corresponding channel.

4.3.4 FlexRay Frame Format

The communication in FlexRay is frame based, i.e. each application message is packed in a *FlexRay Frame* together with other protocol data (see Figure 4-4), before being transmitted on a communication channel. Upon reception, the frame is first unpacked, before the message is stored in a buffer, where it can be read by the application running on the host controller of the FlexRay node. The packing and unpacking of messages is carried out by the communication controller, where the FlexRay protocol is executed.

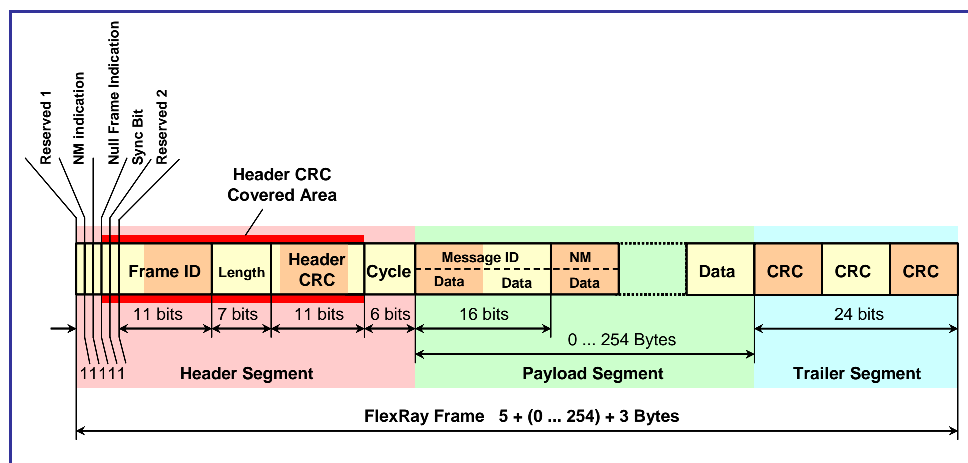

Figure 4-4 FlexRay Frame Format [Flex02]

The most important protocol data within a FlexRay frame is briefly described here in the same order as in Figure 4-4, according to [Flex02].

Frame ID

Each frame has a unique Frame ID which corresponds to a slot number (1 to 2047) within the static segment of a communication cycle. If the frame is intended to be sent during the dynamic segment, then the ID stands for frame's priority.

(Payload) Length

The payload length is the number of 16-bit words in the payload (i.e. the application data). All messages sent during the static segment must have the same payload length.

Header CRC

The header CRC aims at securing the content of the frame header, in order to ensure, that the payload length is correctly received.

Cycle

This is a counter for communication cycles and is therefore synchronously incremented at the beginning of each cycle in all FlexRay nodes.

Payload Segment (Data)

The actual payload is stored in this frame segment, whose length can reach a maximum of 254 bytes.

Trailer Segment (CRC)

The trailer segment of a frame includes an additional 24-bit CRC to validate the integrity of the payload data.

4.3.5 Support for Reliability and Dependability Services

Many implemented mechanisms in FlexRay aim to improve communication reliability and therefore provide an appropriate platform for the efficient implementation of standard protocol-based dependability services, such as the Agreement Protocol service.

By offering the possibility of connecting a node over two communication channels, redundant communication paths can be implemented, so that data transmission is not interrupted, if one communication path fails. As a consequence, the communication medium no longer represents a single point of failure.

Moreover, any faults occurring on the communication path and resulting in the corruption of the exchanged data can be detected by means of the CRC mechanism. As shown in Figure 4-5, however, the error detection in FlexRay covers only the communication between communication controllers. In order to detect a possible node-internal data corruption (e.g. due to memory faults), an additional integrity check mechanism has to be implemented on the application level.

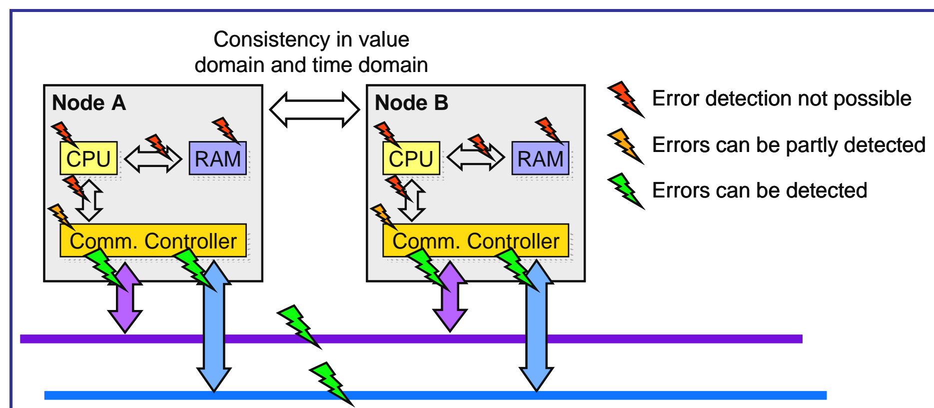

Figure 4-5 Error detection in FlexRay [HSB+02]

Finally, the deterministic and reliable communication scheme offered in FlexRay allows developers to make several assumptions concerning transfer delay and message arrival times. Relying on these assumptions, the implementation of protocol-based dependability services becomes far simpler and more efficient.

5 Agreement Protocols for Safety-Relevant Electronics

To ensure fault-tolerance by means of redundancies in the presence of non-determinism in the domain value, the cooperation of redundant components is required. Malfunctioning components that produce inconsistent information must therefore be handled in such a way, that non-faulty components can reach agreement on a value regardless of faults.

In [LSP80], Lamport et al. provide two different algorithms to solve the problem for redundant, independent nodes. By using only *oral messages* in the first algorithm, at least $n = 3m + 1$ nodes are required to tolerate m faulty ones. This number is reduced to any $n > m$ in the second algorithm which uses *signed messages* instead and tolerates the same number of faulty nodes. Both algorithms allow the non-faulty nodes to compute the same *interactive consistency vector* (*ICV*) containing the private value of each non-faulty node and an agreed value for each faulty node. Once interactive consistency is achieved, each non-faulty node can apply an averaging or filtering function to the interactive consistency vector, according to the needs of the application (subsections 5.1and 5.2).

The *pendulum protocol* presented in [Echt90] aims to reduce the communication overhead in the faultless scenario, so that agreement can be reached with only *n*-1 messages. The protocol always begins with a *main protocol* that will enable agreement, if no faults are present. If the main protocol fails, an *auxiliary protocol* is executed in order to handle the faulty components (subsection 5.3).

Finally, the three algorithms presented, are compared with regard to criteria derived from the requirements of safety-relevant electronic systems (subsection 5.4).

5.1 Oral Messages Algorithm

The notion of oral messages was introduced together with the Byzantine Generals problem in [LSP82]. According to the same source, the definition of an oral message is embodied in the following assumptions with regard to the communication system:

- Every message sent is correctly received.
- The sender of a message is known to the receiver.
- The absence of a message can be detected.

Consequently, the receiver of a forwarded oral message knows who forwarded the message, but has no means to detect whether or not the message has been forged.

In order to tolerate m faulty nodes the oral messages (OM) algorithm requires at least $n = 3m + 1$ nodes and $m + 1$ communication phases. In the first phase, each component communicates its private value (i.e. own value) to every other component. The results of the first phase are then exchanged during the subsequent phases. The non-faulty nodes will forward the values they receive unchanged, while faulty ones may forward falsified values or

even “refuse” to pass information. If a non-faulty node fails to receive a message it expects, it may choose a random value and act as if this value had been received.

An example with $n=4$ nodes is depicted in Figure 5-1 where the only faulty node is B (i.e. $m=1$). The figure shows that all non-faulty nodes (i.e. A, C and D) agree on a same private value for A, whereas the output of the node B is not relevant, since it is faulty. The computation of the private values for C and D is symmetrical to the computation shown for A. If no majority is possible for the computation of the private value of some component, *nil* is recorded.

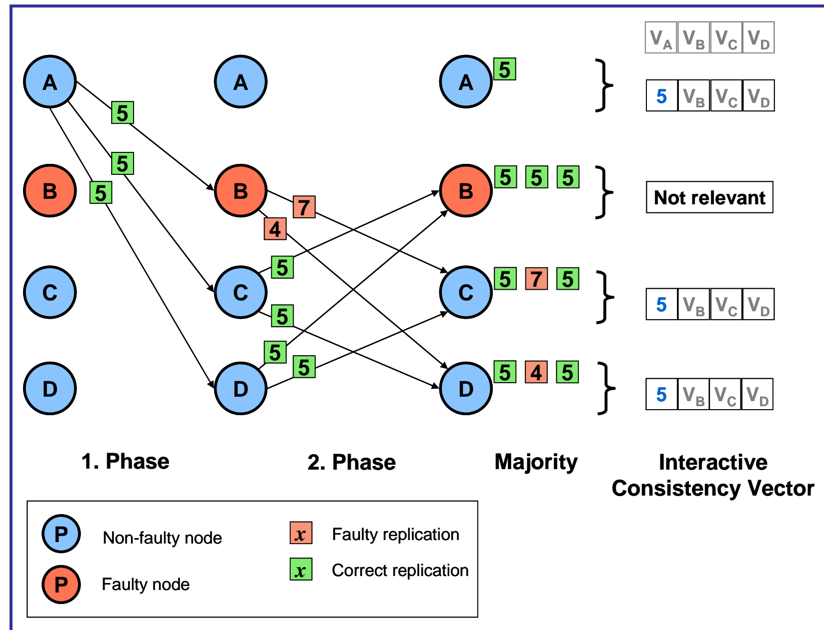

Figure 5-1 OM Algorithm with n = 4 and m = 1

Before describing the general algorithm in detail some terms are defined, as in [LSP80]:

- P is the set of all nodes.
- V is the set of all values.
- A k-level scenario σ is a mapping from a non-empty string of length $\leq k+1$ over P to V, thus summarizing the outcome of a k-phase exchange of messages. For example, let $w=p_1p_2...p_r$. $\sigma(w)=v$ is then interpreted as the value p_2 tells p_1 that p_4 told p_3… that p_r told p_{r-1} is p_r's private value. $\sigma(p)$ simply designates the private value of node p and σ_p is the restriction of σ to the messages received by p in a scenario (i.e. only for $w=pw'$)

In the following, the algorithm OM(m, n, σ) is described in terms of p's computation of the element of the ICV corresponding to each node q, for a given scenario σ, an arbitrary number of faulty nodes $m\geq 0$ and a total number of nodes $n\geq 3m+1$:

(1) If for some subset Q of P of size $>(n+m)/2$ and some value v, $\sigma_p(pwq)=v$ for each string w over Q of length $\leq m$, p records v.

(2) Otherwise, q must be faulty. The algorithm OM($m-1, n-1, \hat{\sigma}$) is then recursively applied with P replaced by $P-\{q\}$, and $\hat{\sigma}$ defined by $\hat{\sigma}(pw)=\sigma(pwq)$ for each string w of length $\leq m$ over $P-\{q\}$. This recursive call delivers an ICV with $n-1$ elements, since the node q is excluded. The i-th element corresponds to the value all non faulty nodes agree it has been received by the i-th node p' from q during the first phase (i.e. $\hat{\sigma}(p')$ which is in fact equal to $\sigma(p'q)$). If at least $\lfloor (n+m)/2 \rfloor$ of the $n-1$ elements agree, then p records the common value. Otherwise, p records *nil*.

The correctness of this algorithm as well as the necessity of $n \geq 3m+1$ are proved in [LSP80].

5.2 Signed Messages Algorithm

Since oral messages can be easily forged by faulty nodes before being forwarded, reaching agreement among non-faulty nodes with the help of the OM algorithm requires at least $n=3m+1$ nodes to tolerate m faulty ones and hence induces a significant communication overhead. By using signed messages instead of oral messages, agreement can be reached for any number $n>m$ nodes and also requires $m+1$ communications phases.

As opposed to oral messages, a signed message from a node p including a data item d cannot be forged, as this message holds an authenticator $A_p[d]$ from p, allowing each receiver to check the message integrity as well as the identity of the original sender. Since this authenticator cannot be falsified, faulty nodes are no longer able to alter the content of a message before forwarding it without being found out. Moreover, forwarded messages are additionally signed by the forwarding node.

Let $v=\sigma(p)$ for a node p (see subsection 5.1 for the definition of σ). p communicates this value to another node r by sending the message consisting of the triple (p,a,v), where a = $A_p[v]$. After receiving the message, r checks whether $a = A_p[v]$. If the test is successful, r takes v as the value of $\sigma(p)$ (i.e. $\sigma(rp)=v$) and forwards the message $(r, A_r[(p,a,v)], (p,a,v))$ to all other nodes except p; otherwise $\sigma(rp)$ = *nil*.

More generally, if r receives a message of the form $(p_1, a_1(p_2, a_2, ..., (p_k, a_k, v)...))$, where $a_k = A_{p_k}[v]$ and for $1 \leq i \leq k-1$, $a_i = A_{p_i}[(p_{i+1}, a_{i+1}, ..., (p_k, a_k, v))]$, then $\sigma(rp_1...p_k)=v$ and r forwards the message after signing it to every other node different than p_1, p_2… and p_k; otherwise, $\sigma(rp_1...p_k)$ = *nil*.

In the following, the signed messages algorithm SM(m, n, σ) is described in terms of the value a non-faulty node p records for a given node q and a scenario σ. m designates the number of faulty nodes ($m \geq 0$) and n the total number of nodes ($n>m$).

Let S_{pq} be the set of all non-nil values $\sigma_p(pwq)$, where w ranges over strings of distinct elements with length $\leq m$ over $P-\{p,q\}$. If S_{pq} has exactly one element v, p records v for q in its ICV; otherwise, q must be faulty and p records *nil*.

As proved in [LSP80], this algorithm allows the non-faulty nodes to compute the same ICV containing the private value of each non-faulty node and *nil* for each faulty node. Although $n > m$ nodes are sufficient to allow non-faulty nodes to reach agreement in the presence of m faulty ones, $n \geq 2m+1$ nodes will be required to enable for example a downstream masker using a majority decision to vote for the correct result.

In [Echt90], this algorithm is slightly modified and presented under the name *distribution protocol* ("Verteilungsprotokoll"). In contrast to the original algorithm, the ICV of each node in the distribution protocol is computed during the communication phases (see Figure 5-2). Before the first phase, each node initializes its ICV to a set of *nil* values, except for its corresponding element where the node's private value is recorded. The vector is then actualized upon each communication phase as follows. When a message $(p_i, a_i(p_j, a_j ... (p_k, a_k, v)...))$ is received by a node p and turns out to be valid, then

- If $ICV_p[k] = nil$, then the value v is recorded for the k-th node, so that $ICV_p[k] = v$.
- If $ICV_p[k] \neq nil$ and $ICV_p[k] \neq v$, then a replication fault obviously occurred in the node p_k and $ICV_p[k]$ is set to *faulty*, so that any further values for p_k are ignored. Nevertheless, valid messages are always forwarded independently of the value of $ICV_p[k]$.

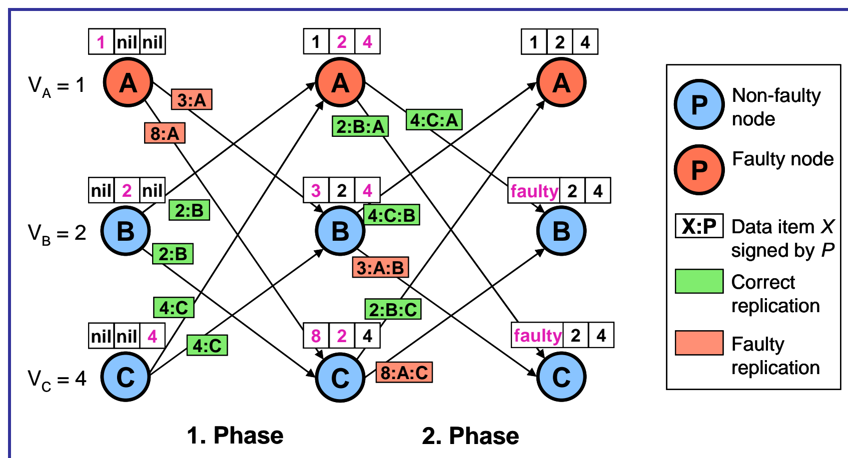

Figure 5-2 Distribution Protocol with 3 Nodes

5.3 Pendulum Protocol

Independently of whether faults actually occur, the protocols previously presented always induce a worst-case communication overhead, since an interactive consistency vector is computed for all nodes before a final result is reached. The *pendulum protocol*, which is introduced in [Echt90], aims to reduce the communication overhead for the faultless scenario at the expense of fault detection by computing a single agreed value. The name of the protocol comes from its algorithm which executes in a similar way to a pendulum stroke over the nodes.

The pendulum consists of a *main protocol*, which is sufficient in the faultless case, and an *auxiliary protocol* applied to deal with faults when these occur. Moreover, it requires the use

of a *distance decision* (i.e. interval decision or sphere decision) and thus the definition of a particular δ (i.e. maximal width of the interval, see subsection 0 for details).

The execution of the main protocol is depicted in Figure 5-3 using the example of three nodes *A*, *B* and *C*, where the node *A* is faulty. Furthermore, the interval decision supports a maximal interval width $\delta = 2$. The node *A* signs its private value $v_A = 11$ and sends it to the next node *B*. If the received message is valid, then *B* compares the received value v_A to its own value $v_B = 12$. Since $|v_A - v_B| < \delta$, *B* records v_A, signs the received message and sends it to *C*. As $|v_A - v_C| > \delta$, v_A fails the relative test of the node *C*. Nevertheless *C* records v_A, since the majority of the nodes (i.e. *A* and *B*) agreed on this value.

Therefore, the non-faulty nodes must outnumber the faulty ones in order to reach agreement on a value that they consistently accept. The example shows that even the value computed by a faulty node might be agreed on as long as this value belongs to the same interval of maximal width δ as at least the half of the other nodes. For this particular reason, the pendulum protocol is not suited for applications with high requirements for accuracy and correctness.

Moreover, it is shown, that if successful, the main protocol needs only $n-1$ messages to reach agreement among *n* nodes.

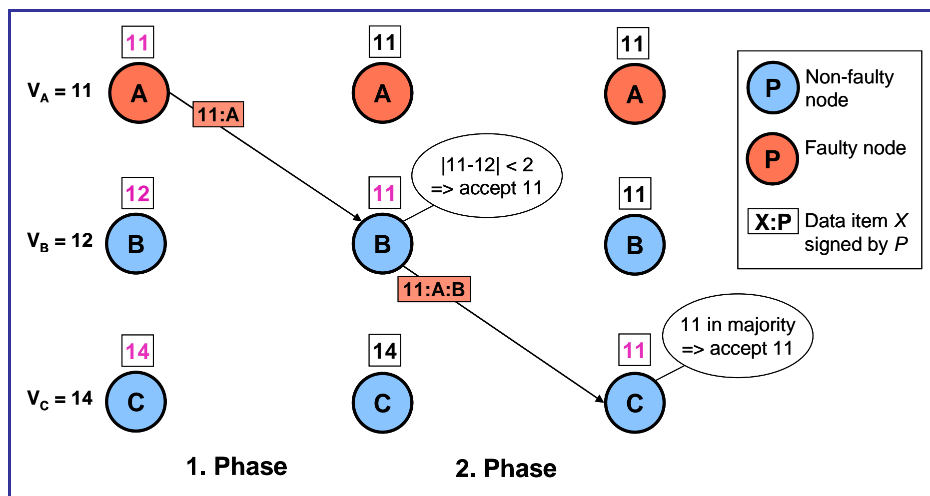

Figure 5-3 Main Protocol (pendulum protocol) for 3 nodes, $\delta = 2$

The main protocol fails due to a node *p* if this node is faulty and refuses to send messages, or if the propagated value is still not agreed upon by the majority of the nodes and additionally does not pass the relative test of *p* which might be non-faulty. In both cases, *p* will not generate any message and the pendulum stroke is interrupted. This causes the subsequent node to time out and hence to detect the occurrence of a fault, upon which the auxiliary protocol is applied. A simple way to implement the auxiliary protocol is to choose another agreement protocol (e.g. SM algorithm) to be executed only when the main protocol fails. In this case, the overall communication overhead will be greater than simply applying the chosen agreement protocol from the beginning. Alternatively, further strokes, originating in other nodes, might be carried out in order to reach agreement on another node's value. The

communication overhead produced in this case cannot be predicted exactly, as it depends on the number of strokes preceding a final stroke that will lead to agreement.

5.4 Comparison and Conclusion

In the previous subsections, three different agreement protocols have been discussed. These are compared here with regard to the following criteria:

- Predictability: the algorithm behavior does not depend on run time events (e.g. the occurrence of faults).
- Fault-tolerance: agreement can be reached despite the presence of faults.
- Fault-detection: the algorithm includes mechanisms that enable the detection of components providing faulty values.
- Number of redundancies: the minimal number of redundancies required to tolerate m faulty processing units.
- Minimal communication overhead: the minimal number of messages required to reach agreement among n processing units, e.g. in the absence of faults.
- Maximum communication overhead: the maximal number of messages required to reach agreement among n processing units, e.g. in the presence of faults.
- Number of communication phases: the number of communication phases required to tolerate m faulty processing units.

These criteria can be classified with respect to their priorities, as depicted in Table 5-1. This classification results from the stringent requirements defined in subsection 2.2.2 as well as the fact that efficiency is not a primary requirement in the scope of this work.

Very high	High	Medium
▪ Predictability ▪ Fault tolerance ▪ Fault detection	▪ # of redundancies	▪ Minimal com. overhead ▪ Maximum com. overhead ▪ # of com. phases

Table 5-1 Criteria Priorities

Based on these criteria, the three agreement protocols are compared in Table 5-2, where n is the total number of processing units and m the number of faulty units to be tolerated.

From the results of this comparison, the pendulum protocol does not appear to be adequate, even if it provides the lowest communication overhead in the faultless scenario. Besides failing to fulfill the predictability requirement, it does not provide fault detection and could even lead to the agreement on a faulty value (see example in subsection 5.3). Moreover, the pendulum protocol is limited to applications supporting only an interval or sphere decision, which makes it less qualified for a standard service implementation.

	OM Algorithm	SM Algorithm	Pendulum Protocol
Predictability	Yes	Yes	no
Fault tolerance	Yes	Yes	yes
Fault detection	Yes	Yes	no
# of redundancies	$3m+1$	$>m$	$2m+1$
Minimal com. overhead	$\frac{n\cdot(n-1)\cdot((n-1)^{m+1}-1)}{n-2}$	$\frac{n!}{(n-2)!}+\ldots+\frac{n!}{(n-m-2)!}$	$n-1$
Maximal com. overhead	$\frac{n\cdot(n-1)\cdot((n-1)^{m+1}-1)}{n-2}$	$\frac{n!}{(n-2)!}+\ldots+\frac{n!}{(n-m-2)!}$	variable
# of com. phases	$m+1$	$m+1$	variable

Table 5-2 Comparison of three Agreement Protocols

Both the OM and the SM algorithms fulfill the most relevant requirements. Although the signing of messages induces an additional computational overhead, the SM protocol is most appropriate in the scope of this work, since it can guarantee fault-tolerance with less redundancies and a lower communication overhead than the OM protocol. Therefore, the concept presented in the next subsection will be based on the SM algorithm.

6 Concept for an Agreement Protocol Service

After a thorough analysis of three kinds of agreement protocols, these variants have been compared with respect to a set of relevant criteria derived from the requirements for safety-related services (see subsection 5.4). The outcome of this comparison, depicted in Table 5-2, shows that the SM protocol is most appropriate for implementation as a standard service, called Agreement Protocol service (AP service), in the scope of the EASIS architecture.

Before considering the conception of the AP service the requirements and constraints that have to be taken into account for its specification and design are discussed (subsection 6.1).

The specification of the AP service follows a top down approach. Firstly, the service will be considered as a black box in order to identify potential interactions with its environment based on the derived requirements and constraints, without getting into internal implementation details (subsection 6.2).

In a further step, the internals of this black box are specified in the form of several units that interact with each other to compute the desired output in compliance with the functional requirements of the AP service (subsection 6.3).

Some application examples of the AP service, concluding this section, demonstrate how the AP service can be used within an application (subsection 6.4).

6.1 Requirements and Constraints of the AP Service

Considering the particular constraints of this work, the SM protocol can not be implemented in its original form. It must be tailored to the specific requirements for the Agreement Protocol service (AP service) in EASIS, to the constraints resulting from its integration in this architecture and from the nature of potential applications using this service. Consequently, it is necessary to first derive these requirements and constraints in order to elaborate an adequate concept for the AP service based on the SM protocol.

6.1.1 Requirements to the AP Service

The main requirement for the AP service is the computation of an *agreed value* among the participating nodes from a set of initial values (i.e. an initial value for each node) stemming from local application components. The AP service must ensure that the non-faulty nodes will compute the same agreed value and thus will exhibit the same behavior. Subsection 6.4 includes examples on how this output can be used within the scope of an application.

Besides this primary functionality, which represents the integral part of the SM protocol, the AP service to be embedded in EASIS must provide additional information regarding potential fault sources, in the case of discrepancies between the initial values. This information can be gained by extending the SM protocol properly and will enable fault diagnosis, so that faulty nodes can be detected. This information will be presented in the form of a *state vector* including the state of each participating node, from the point of view of the node from which this vector is generated.

Finally, the agreement protocol functionality in the EASIS architecture is intended to be provided as a standard service. This requires the specification of a well defined interface not only to the application layer, including the appropriate input and output ports through which the AP service can be accessed and executed, but also to the other basic software modules.

6.1.2 Requirements of the AP Service

The requirements of the AP service are mainly derived from those of the SM protocol, since the latter constitutes the basis for the former.

First, the protocol assumes the presence of a communication system to ensure the transmission of messages between the participating nodes. As stated in subsection 5.1, the communication system must fulfill the following requirements:

- Every message sent is correctly received.
- The sender of a message is known to the receiver.
- The absence of a message can be detected.

Bearing these assumptions in mind, the SM protocol has to be adapted to the communication services available in the EASIS architecture, so that these can be used by the AP service without affecting the functionality of the protocol.

Furthermore, the SM protocol does not assume anything about the number of nodes to ensure agreement among the non-faulty ones. As a consequence of a signature mechanism, any number of non-faulty nodes can agree on the same ICV in the presence of any number m of faulty nodes, if sufficient (i.e. $m+1$) communication phases are executed (see subsection 5.2). The total number of nodes depends only on the application and its fault assumption. Consequently, the AP service should not have any requirements concerning this, since it must be independent from the application. However, the use of a signature mechanism implies the requirement for specific information (e.g. keys, node IDs) in order to generate and check signatures. This information must be provided by the application since it depends both on the nodes executing the agreement protocol and on their number.

In contrast to the SM protocol, the AP service must provide a final agreed value and not an ICV. Therefore, a decision mechanism is also required in order to compute this value from the ICV. The choice of a particular decision function is not relevant. It is, however, crucial that the nodes, among which an agreement protocol is executed, implement the same decision function to ensure that the same agreed value is computed.

6.1.3 Constraints

The integration of the AP service in the EASIS architecture and the nature of the potential applications that will use the protocol give rise to a number of constraints that have to be taken into account for the conception of the service.

As described in subsection 2.4.3, the EASIS Architecture includes several basic services that must be exploited, whenever possible, since this guarantees that no function is implemented redundantly and reduces the use of non standard functions. Depending on the requirements

described in subsection 6.1.2, the basic services that can be used, as well as their interfaces, must be identified, which in turn will determine some of the interfaces of the AP service.

As already discussed in subsection 5.2, the SM protocol is based on a signature mechanism. By using signatures, the receiver of a message can check the identity of the sender and hence the authenticity of the received message. The complexity of the signature depends on the nature of the attacks and faults that might be encountered, i.e. on the nature of the application and the runtime environment. As opposed to applications in the IT domain, automotive applications are not subject to malicious attacks, but only to hardware or software faults that might lead to a Byzantine behavior, such as forwarding a correctly received value to the wrong node. Moreover, the available resources for these applications are very limited. Therefore, a cryptographically strong signature mechanism for the AP service is not required. For example, the use of encryption or hash functions is not necessary. A simpler but sufficient mechanism will be presented in subsection 6.3.1.

Finally, the execution of a communication phase in a node assumes that all the messages sent during the previous phase have been received. In a non-deterministic environment, therefore, each node has to keep count of the number of the messages received and use a timeout to stop waiting for messages that have been lost or have not been sent. Since safety-related systems rely on a time-triggered architecture, the behavior of the operating system (i.e. the execution of tasks) and the communication system (i.e. the transmission of messages) is deterministic. As a consequence, the detection of a missing message does not require waiting nor a timeout mechanism, since the arrival time of each message is known in advance. Moreover, the synchronization of the different communication phases between the nodes participating at the protocol is inherent in the underlying time-triggered operating system.

6.2 Integration in the EASIS Architecture

Based on the requirements analyzed in the previous subsection, the potential interactions between the AP service and its environment are identified in this subsection. The EASIS software architecture represents the environment, where the AP service is to be embedded. As already mentioned in subsection 2.4.3, it provides a set of dependability services such as a Fault-Tolerant Communication (FTCom) service that will be used by the AP service to ensure the sending and receiving of messages between the nodes executing the protocol, which gives rise to the interface depicted in Figure 6-1.

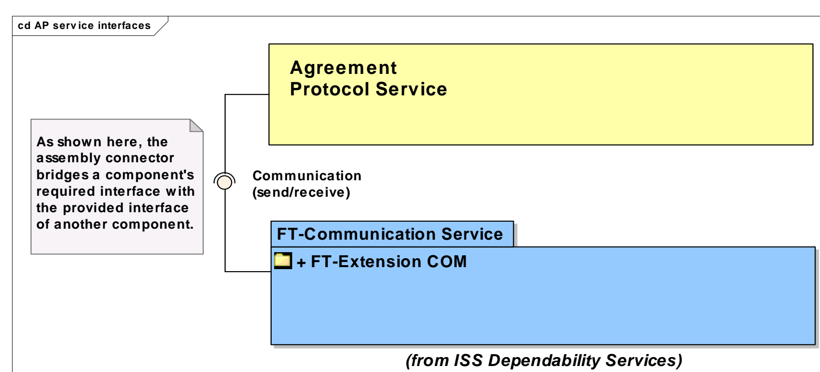

Figure 6-1 Interface to FTCom

Excluding FTCom, no other service from EASIS is required. The next step is, therefore, to determine the interaction of the AP service with the application layer.

The application using the AP service must provide an input value as the private value of the node, the number of phases to be executed, the tolerance band and additional node information required for the signature mechanism. The tolerance band is used to support applications providing non-deterministic input values, such as sensor values.

Upon its execution, the AP service delivers both a final agreed value and a state vector to the application. The resulting interface is depicted in Figure 6-2.

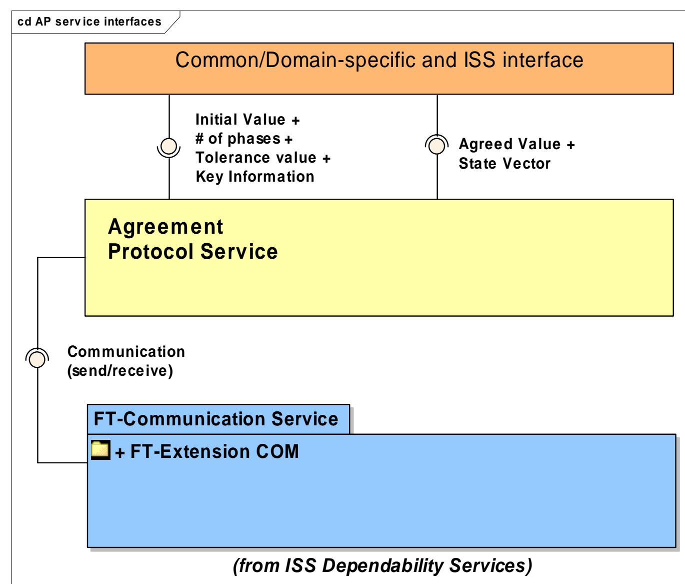

Figure 6-2 Interface of the AP service

6.3 Internal Design of the AP Service

After having specified the interface for the AP service, the internal design of the service is presented here. In addition, the behavior of the AP service in the presence of a communication failure will be also discussed.

As described in subsection 6.1, the AP service includes both a signature mechanism and a decision mechanism. Additionally, a main core is required, which takes care of the execution of the actual masking protocol (i.e. the agreement protocol) and the computation of the output values. Some characteristics of the signature mechanism (e.g. length of the signature) and the decision mechanism (e.g. the decision function or the handling of error values) may depend on the application. However, these mechanisms are not provided as basic services in the scope of the EASIS architecture. Therefore, they are integrated in the AP service as sub-services which are implemented separately from the main core, to allow for flexibility in the implementation of the AP service with different applications. These sub-services are called *signing unit* and *decision unit*, while the main core using them is called *masking unit*, as depicted in Figure 6-3.

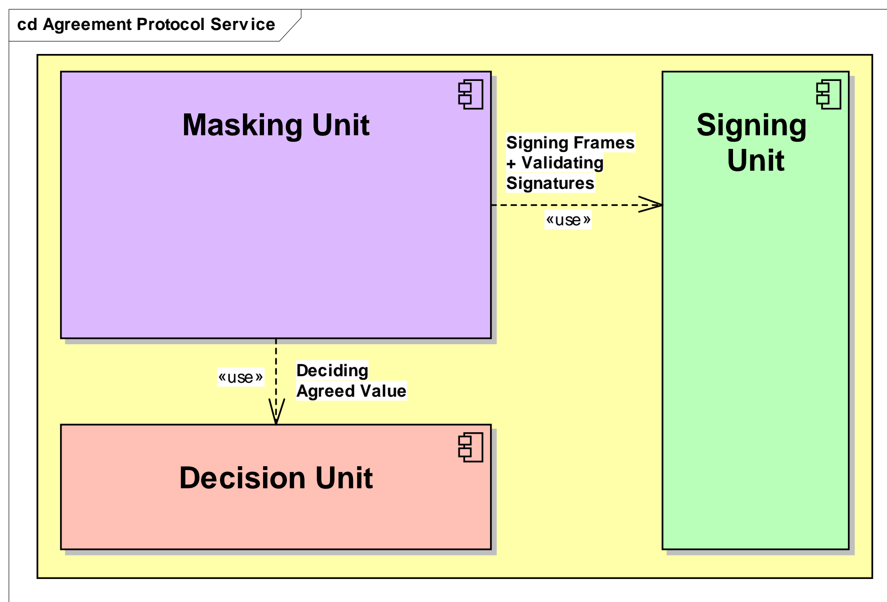

Figure 6-3 Different units constituting the AP service

Before detailing the specification of these units, the notion of AP frames must be presented for the better understanding of the following subsections. Since the values exchanged in the scope of the masking protocol must be signed, a message which is exchanged between two different nodes consists of more than a single value. Therefore, frames are required where value, signature and other protocol information can be packed together. These frames are called AP frames and consist of three fields, as depicted in Figure 6-4.

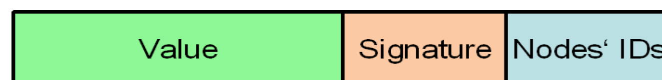

Figure 6-4 AP Frame

Value

This field contains the actual value to be exchanged.

Signature

Each value exchanged in the scope of the agreement protocol must be signed from one or more nodes. This field contains, therefore, the signature information.

Nodes' IDs

This field contains the IDs of all the nodes that have seen and signed the value included in the AP frame. All IDs are encoded in the same number of bits, which simplifies their extraction. The number of IDs in this field can not exceed the number of phases required for the protocol, as explained in subsection 6.3.3.

In order to simplify the use of AP frames in the next subsections, functions for generating frames, changing them and extracting specific information are defined as follows:

generateFrame(value)

This function creates a new frame including the specified value with an empty signature field and an empty IDs field.

getValue(frame)
This function extracts the value included in a frame.

getSignature(frame)
This function extracts the signature included in a frame.

setSignature(frame, new_signature)
This function sets the signature field of a frame to the specified new signature.

getIDs(frame)
This function extracts the IDs list from a frame.

appendID(frame, ID)
This function appends a new node ID to the IDs field of a frame. This function is required, when a signed value is co-signed by another node.

6.3.1 The Signing Unit

The signing unit must provide two functionalities to the masking unit: the generation of signatures for AP frames and their validation. This leads to the UML diagram depicted in Figure 6-5. The unit is modeled as an interface, in order to indicate that different implementations are possible, although a concept for a signature mechanism will be presented here. As discussed in subsection 6.1.3, a cryptographically strong mechanism for the generation of signatures is not necessary. Therefore, simple and efficient algorithms are used instead, which are based on the computation of a CRC for the value included in the AP frame.

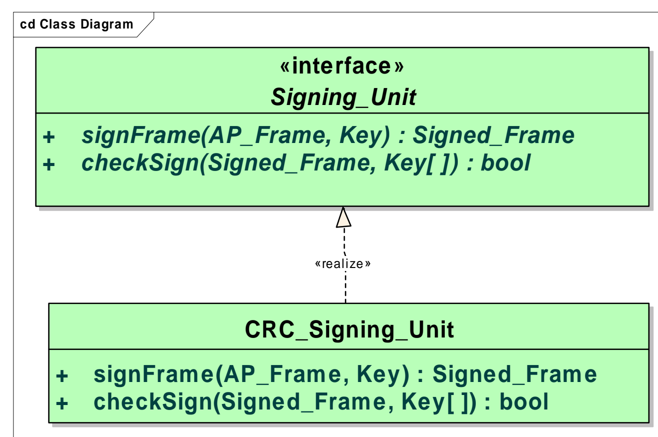

Figure 6-5 Signing Unit

Since each node participating in the agreement protocol must sign a frame before sending it or forwarding it, an AP frame might contain multiple signatures. For a better understanding of the concept, the algorithms will first be described in terms of a single signature. The handling of multiple signatures will be left for the section concerning implementation.

The presented mechanism requires a private key *rK* and two public keys *uK1* and *uK2* for each node. *rK* and *uK1* can be chosen randomly and should preferably be of the same length, while *uK2* is computed as follows:

$$uK2 = rK \oplus uK1 \text{, where } \oplus \text{ is a bitwise XOR operator}$$

The private key is kept secret by the node generating the signature, while the public keys are made available to the other nodes wishing to receive and validate signed frames from the first node. All this information (i.e. the local private key and the public keys of all the nodes participating in the agreement protocol) must be provided to the AP service and thus to the signing unit by the application.

It is clear that the private key of a node can be easily computed from the public keys of this node. This would require, however, a targeted and malicious intervention, which is not given in automotive applications.

The signature for a frame *frame* can be computed in a node with the private key *rK* by the algorithm signFrame (*frame*, *rK*) as follows:

```
value = getValue(frame);
crcValue = computeCRC(value);
signature = crcValue ⊕ rK;
setSignature(frame, signature);
```

From this algorithm, the following equation results:

$$signature \oplus uK1 = (crcValue \oplus rK) \oplus uK1 = crcValue \oplus uK2$$

Using this key idea, any node receiving the signed frame can check the validity of the signature by using the algorithm checkSignature (*frame*, *uK1*, *uK2*), assuming it knows the public keys *uK1* and *uK2* of the node where the signature is generated:

```
value = getValue(frame);
crcValue = computeCRC(value);
signature = getSignature(frame);
if (crcValue ⊕ uK2) = (signature ⊕ uK1) then
    signature is valid;
else
    signature is not valid;
```

6.3.2 The Decision Unit

The decision unit must provide the function of deciding a value from a vector of values, as shown in Figure 6-6. In the scope of the AP service, the decided value represents the agreed value that must be delivered by the service, while the vector of values is the ICV computed by the masking unit. The decision unit is modeled as an interface for the same reason as the signing unit. While the implementation of decision functions should be straightforward, special care has to be taken to handle error values that might be generated by the masking unit, as described in the implementation section.

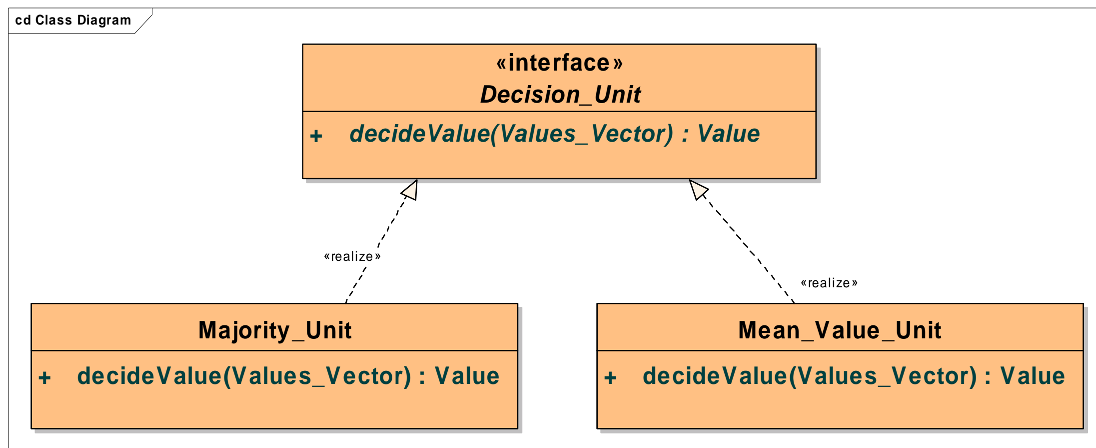

Figure 6-6 Decision Unit

If a decision is not possible (e.g. no majority when using a majority decision), a *NIL* value will be returned by the AP service as the agreed value. In this case, the application will recognize that the number of faulty nodes exceeded the number specified in the fault assumption made for the application. It can then use a default value, initiate a safe shut down of the system or restart the application.

6.3.3 The Masking Unit

The masking unit represents the core of the AP service, where the actual masking protocol is executed and the final output values are computed. The concept for the masking protocol is based on the SM algorithm described in subsection 5.2. The protocol executes in phases, where the private value of a node is sent to all other nodes during the first phase. The received values are then exchanged among all nodes in subsequent phases until agreement is reached. The total number of phases is determined by the number of faults to be tolerated, i.e. $m+1$ phases are required to tolerate m faulty nodes. Besides the computation of an agreed value, the masking unit must provide a state vector, which leads to the class diagram in Figure 6-7.

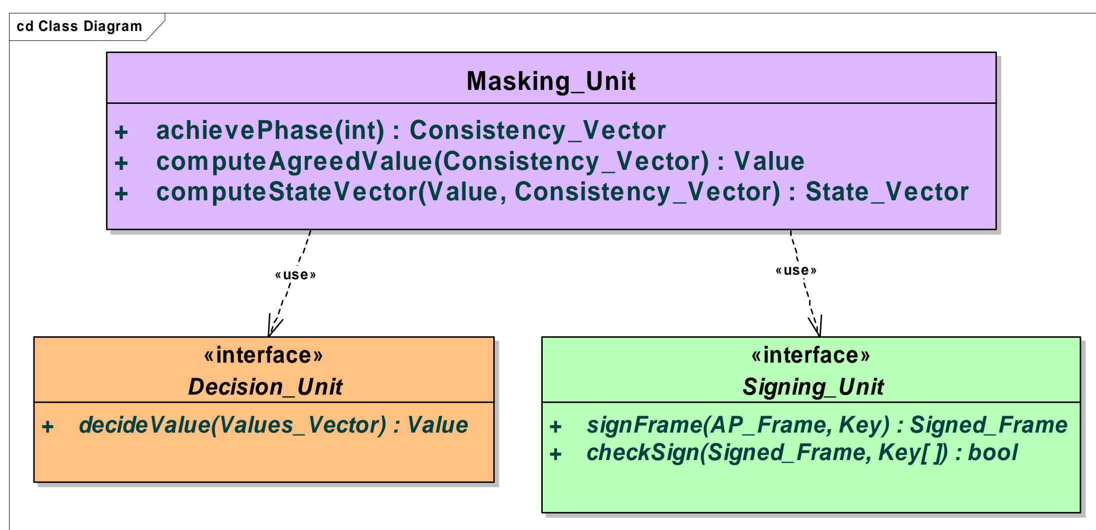

Figure 6-7 Masking Unit

The algorithm achievePhase (1) for the first phase is given for a node with the ID *nodeID* and the private value *privVal* as following, where *ICV* designates the ICV to be computed by the node:

```
frame = generateFrame(privVal);
sign frame;
appendID(frame, nodeID);
send frame to all other nodes;
set own value in ICV to privVal and store NIL for all other nodes;
```

The node ID of each node must be appended to the signed value before sending or forwarding a frame, since information concerning the original sender and intermediate nodes, required for signature validation, is not necessarily provided by the communication service. Therefore, a new node ID is appended to an AP frame at each phase, which brings the total number of IDs stored in the frame to the number of phases to be executed.

Furthermore, the IDs field can be used to identify the nodes that have not yet seen the value, as in the algorithm achievePhase (*i*), where subsequent phases are executed:

```
frames_set = frames received in phase i-1;
for each frame in frames_set
{
    if signature is not valid go on to next frame;
    sign frame;
    appendID(frame, nodeID);
    ID_list = getIDs(frame);
    send frame to any node whose ID is not in ID_list;
    origID = node ID of original sender of the frame;
    val = getValue(frame);
    if (ICV[origID] = NIL) or (ICV[origID] = val) then
        ICV[origID] = val;
    else ICV[origID] = FAULT;
}
```

The content of the ICV is updated at the end of each phase, as in the distribution protocol variant of the SM protocol (see subsection 5.2). Moreover, the algorithm implies that the value in the ICV corresponding to a node will remain equal to *NIL,* for as long as frames with the private value of this node are not received. Consequently, the values that might be included in the ICV of a node are private values, *NIL* entries or *FAULT* entries, where *FAULT* is the value stored for a node which issues inconsistent private values. The final ICV is then input to the decision unit, where the agreed value is computed.

Besides the agreed value, a state vector (SV) must be computed, including the state for each node participating at the agreement protocol from a local point of view. Three basic states are possible for a node: *CORRECT, INCORRECT* and *FAULTY*, which are defined as following:

CORRECT: the ICV value corresponding to the node (i.e. the private value of the node) is equal to the computed agreed value.

INCORRECT: the ICV value corresponding to the node is different from the computed agreed value.

FAULTY: the node delivered inconsistent private values.

However, the available protocol information might not always be sufficient to determine the state of a node. For example, it is not correct to presume that a node is faulty if no information is received from or about this node due to a failure of communication with this node. The same is true when the computed agreed value is equal to *NIL*. Therefore, a fourth state is required, the *UNKNOWN* state.

Upon execution of the last communication phase, the final ICV is computed by the algorithm computeFinalCV() as follows:

```
frames_set = frames received in last phase;
for each frame in frames_set
{
    if signature is not valid go on with next frame;
    origID = node ID of original sender of the frame;
    val = getValue(frame);
    if (ICV[origID] = NIL) or (ICV[origID] = val) then
        ICV[origID] = val;
    else ICV[origID] = FAULT;
}
```

The algorithm computeStateVector (*agreedVal, ICV*) that computes the SV using the agreed value *agreedVal* and the final ICV *ICV* is given here:

```
for each nodeID of all participating nodes
{
    if (agreedVal = NIL) then
        SV[nodeID] = UNKNOWN;
    else
    {
        val = ICV[nodeID];
        if (val = agreedVal) then SV[nodeID] = CORRECT;
        else if (val = NIL) then SV[nodeID] = UNKNOWN;
```

```
            else if (val = FAULT) then SV[nodeID] = FAULTY;
            else SV[nodeID] = INCORRECT;
        }
}
```

Since some applications would consider slight deviations in the values as acceptable (e.g. if sensor values are used), a private value diverging from the agreed value should not necessarily result in the *INCORRECT* state. Therefore the algorithm has to be extended to support such applications. This is done by replacing the equality check in the algorithm by the following check:

$$\text{if } |val - agreedVal| \leq \delta \text{ then } SV[nodeID] = CORRECT,$$

where δ designates the tolerated deviation which is provided by the application. The new algorithm is:

```
for each nodeID of all participating nodes
{
        if (agreedVal = NIL) then
            SV[nodeID] = UNKNOWN;
        else
        {
            val = ICV[nodeID];
            if |val – agreedVal| ≤ δ then SV[nodeID] = CORRECT;
            else if (val = NIL) then SV[nodeID] = UNKNOWN;
            else if (val = FAULT) then SV[nodeID] = FAULTY;
            else SV[nodeID] = INCORRECT;
        }
}
```

6.3.4 The Impact of Communication Failures

The core of the AP service is a masking protocol which requires a reliable communication in order to guarantee a fault-tolerant agreement among non-faulty nodes. More specifically the following is assumed:

- Every message sent is correctly received.
- The sender of a message is known to the receiver.
- The absence of a message can be detected.

The concept of the AP service presented here relies on the FTCom service for communication. Even though this service is fault-tolerant, it can not guarantee the first condition, i.e. every message sent is correctly received.

In fact, should the entire underlying communication system fail, then no messages can be transferred and thus the execution of a protocol would be meaningless. Each node executing the AP service will only know their own local value and will store a series of NIL values for the other nodes. This must be handled by implementing the decision unit properly to meet the application requirements in this case, for instance, by delivering a default value.

Even if a retransmission mechanism is provided, transient or intermittent faults in the communication system might also affect the functionality of the AP service in a real-time system as they can lead to a timing failure, i.e. the late delivery of frames in this case, causing these frames to be ignored. However these kinds of faults are less serious than a total failure of the communication system since they can be tolerated under specific assumptions. The nature of the network topology plays an important role and that is why we will distinguish between a bus topology and point-to-point connections.

Bus Topology

If all the nodes executing the AP service communicate over a bus, then the loss of a frame would cause all the nodes not to receive this frame. As a consequence, a transient communication fault during the first phase affecting at least one frame might lead to inconsistent agreed values, even if all the nodes are non-faulty. In fact, the node, whose frame has not been transmitted correctly, has a value more (i.e. the own private value) in its ICV than the other nodes, even upon execution of subsequent communication phases.

Point-to-Point Connections

In the case of point-to-point connections, where each pair of nodes are connected over a separate path, a communication failure during the first phase can be tolerated if each node can reach at least one other node and if the second phase executes without failures. This will guarantee that the private value of the first node is propagated to all other nodes during the second phase by the node reached in the first phase, so that all the nodes store the same set of values upon execution of the protocol.

6.4 Examples

In order to demonstrate the operation of the AP Service, the three nodes *A*, *B* and *C*, as depicted in Figure 6-8 will be considered. The AP service of each of these nodes obtains, besides service-relevant information, a private value from the application interface. It is further assumed that at most one node can be faulty. Assuming this fault occurs effectively in the node *A* (marked in red color), this node will propagate inconsistent information to the rest of the nodes.

The protocol executes as described in the algorithms above, relying on the FTCom service for the transmission of AP frames. Upon execution, an ICV is computed in each node. The computation of an agreed value and an SV is shown in Figure 6-8 only for the node *B*. The agreed value is computed from the ICV by applying the interval decision (see subsection 0). Both this value and the ICV are then used to compute the SV. Finally the AP service provides the computed information to the application interface.

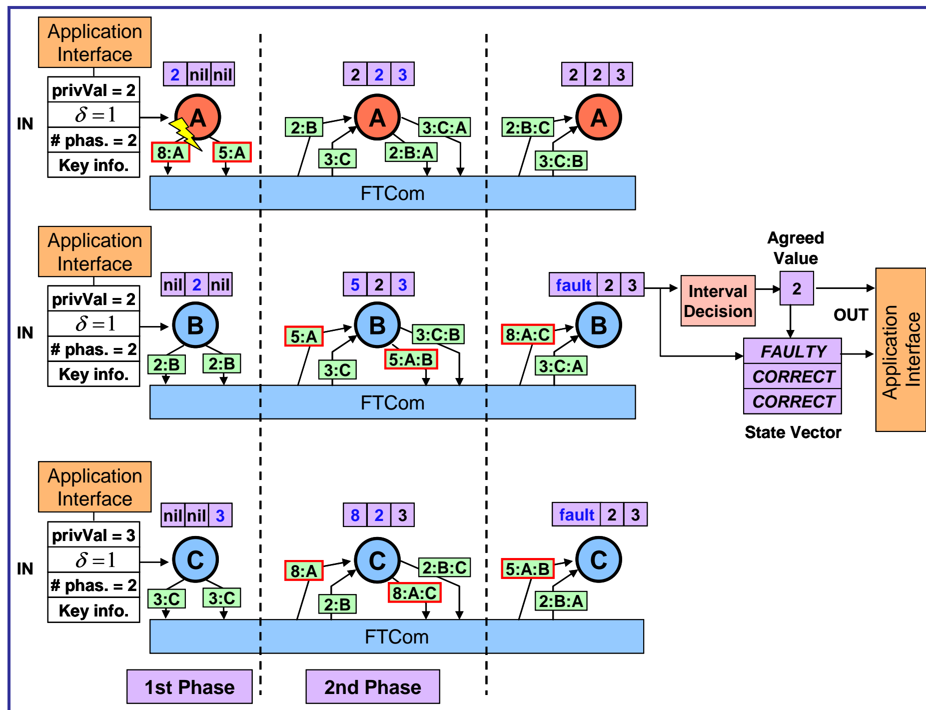

Figure 6-8 Example of the AP service execution

The usage of the AP service implies the existence of redundant information. The purpose of using an agreement protocol is not to get rid of the redundancy, but to make sure it is consistent, even in the presence of non-determinism and faults. Upon execution of the protocol, the application can continue processing redundant values (i.e. the computed agreed values) simultaneously, as shown in Figure 6-9, so that redundant output values are computed in the scope of subsequent application components (corresponding to the application component *B* in Figure 6-9). The interrogation marks in the output of the node *A* indicate that the output of this node is not relevant since the protocol detected the occurrence of faults in this node.

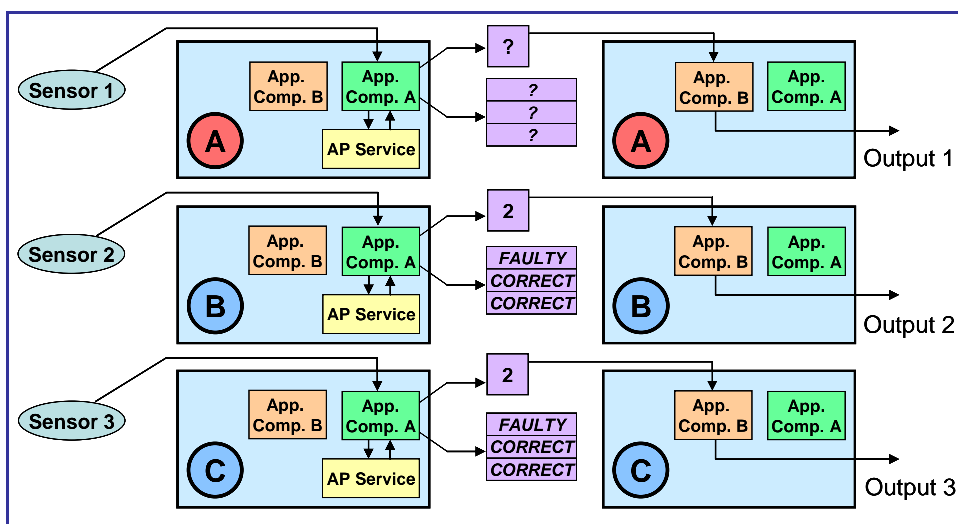

Figure 6-9 Simultaneous use of agreed values

Another alternative would be to compute a single value from the redundant agreed values (e.g. by using a majority voter as in Figure 6-10). This value, assumed to be the correct value,

is then processed by other application components, so that a single output value is finally produced.

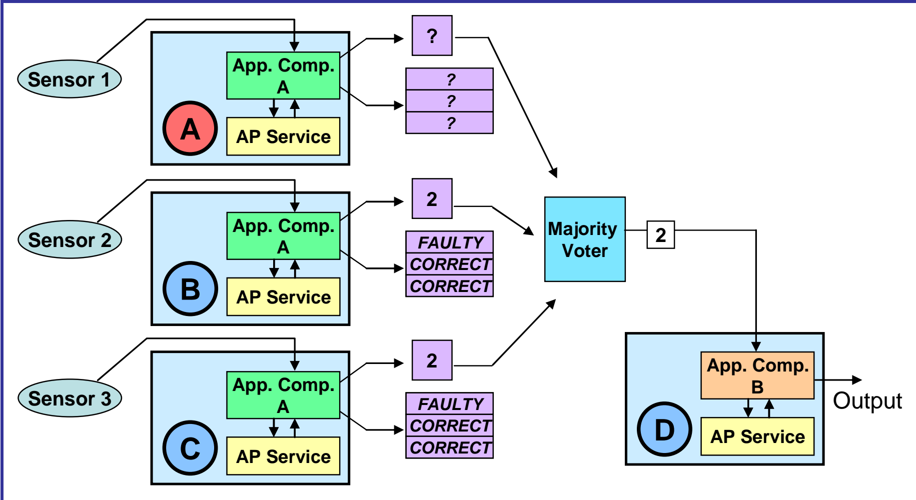

Figure 6-10 Computation of a single agreed value

7 Implementation of a Virtual Prototype

For the validation of the AP service, a virtual prototype was implemented in Matlab/Simulink in accordance with the concept presented in section 6.

Firstly, the requirements for the prototype, as well as the constraints that might affect its implementation, are discussed in subsection 7.1.

Further, the implementation tools used for the realization of the prototype are presented. These consist mainly of the development environment Matlab/Simulink, as well as third party tools and block sets (subsection 7.2).

The implemented prototype represents a system consisting of n redundant nodes. Each node has a private value, received from an emulated sensor, and runs an AP service which ensures that all the non-faulty nodes use the same value. The implementation of the AP service is described in details in subsection 7.3.

7.1 Requirement and Constraints

The main requirement for the virtual prototype implemented for the validation of the AP Service, is compliance with the specification and UML design presented in section 6. Hence, the implemented AP Service must have all the specified interfaces and exhibit an internal design in the form of three units (i.e. masking unit, signing unit and decision unit), such that every implemented function can be unambiguously assigned to one of these units.

Moreover, the AP service must be implemented in such a way, that application-specific parameters, such as the number of nodes, the number of communication phases or the size of the AP frame, can be easily configured in the Simulink model without affecting the internal implementation of the service functionalities.

Besides these requirements, certain constraints must also be taken into consideration for the implementation of the prototype.

As described in subsection 6.3.3, the execution of the AP service in a node is sequential and consists of different communication phases. Since the results of each phase (e.g. the updated ICV) must be available for the subsequent phase and the computation of these results involves all nodes, a synchronization effort between the nodes is required. Therefore, the different phases are implemented as tasks, which are dispatched in each node according to a uniform schedule, ensuring that similar tasks are executed simultaneously among all nodes. This approach requires a time-triggered operating system where the time-triggered scheduling and dispatching of tasks is inherently implemented (see subsection 4.2). Moreover, this would enable a predictable overall execution of the AP service, so that stringent execution deadlines, which are typical to real-time safety-critical systems, are not exceeded.

In order to support the time-triggered execution of communication phases, the exchanged AP frames must always reach their targets before the next phase begins, independent of the communication load. This can be guaranteed if a time-triggered communication system is

used (see subsection 4.3). Therefore, the prototype implemented relies on a simulation for the communication system FlexRay that ensures the exchange of AP frames between system nodes.

7.2 Implementation tools

The virtual prototype constitutes a system model which is designed in the Matlab/Simulink development environment. Specific functionalities that are not provided in Simulink standard libraries are implemented by means of S-Functions.

Besides Simulink standard blocks and self-implemented blocks, customized block-sets provided by the company DECOMSYS are used to simulate the execution environment as well as the communication system. The generation of a communication schedule for the FlexRay communication system requires the use of the tool DESIGNER, also by DECOMSYS.

7.2.1 Matlab/Simulink Standard Blocks

Matlab/Simulink offers a wide range of standard blocks. However, only some of these blocks are used for the design of the virtual prototype, since most of the required functionalities, such as the execution of communication phases or the computation of signatures, are very specific. The following standard blocks are used in the modeling of the virtual prototype:

Constant Block

The Constant block, depicted in Figure 7-1, is used to output a constant, specified in its configuration. This constant can be a single value, a 1-Dimensional vector or even a matrix.

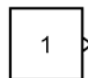

Figure 7-1 Constant Block

Mux and Demux Blocks

The Mux and Demux blocks shown in Figure 7-2 belong to the signal routing library of Simulink. While the Mux block multiplexes signals into a single bus, the Demux block splits a bundled signal into its constituent scalar or vector signals.

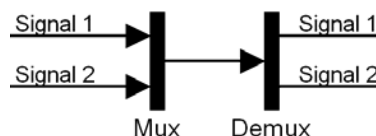

Figure 7-2 Mux and Demux Blocks

Goto and From Blocks

The Goto and From blocks are used to reduce the number of connections in the model and thus to preserve a simple overview. When a value (e.g. a constant) is assigned to a Goto block with a specific tag, all From blocks carrying the same tag will read this value, so that the routing of this value to the subsystems or blocks using it can be avoided (see Figure 7-3).

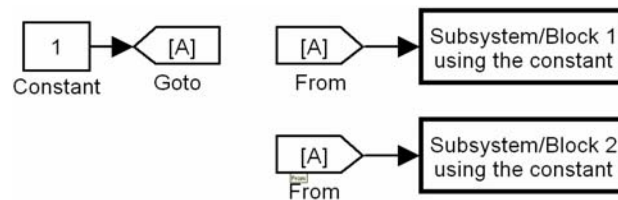

Figure 7-3 Goto and From Blocks

In and Out Blocks

When added to a subsystem, an In or Out block (shown in Figure 7-4) specifies a new input or output port for this subsystem.

Figure 7-4 In and Out Blocks

Display Block

The Display block (see Figure 7-5) is used to numerically display the input value. It is merely used as a visual instrument of verification.

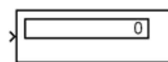

Figure 7-5 Display Block

7.2.2 S-Functions

For the functions that are more specific to the AP Service, S-Functions must be implemented in order to describe the behavior of user-defined blocks (also called subsystems in this case). While Simulink offers the possibility to write S-Functions in ADA, C or C++, C is used in the scope of this work to ensure compatibility of the virtual prototype with tools used for subsequent integration steps.

An S-Function block is defined by a C MEX-file that provides information about the block to Simulink during the simulation, such as initial conditions, block characteristics or outputs. Simulink interacts with a C MEX-file S-function by invoking the callback methods that the S-function implements. Each method performs a predefined task (e.g. computing block outputs) required to simulate the block. By implementing these methods, it is possible to create custom blocks with the desired behavior. The two callback methods which are relevant in the scope of this work are the following:

void mdlInitializeSizes(param)

Simulink calls this method to inquire about the number of input and output ports, sizes and types of the ports, and any other objects, such as the number of states, needed by the S-function. The ports of an S-Function block can be dynamically sized, so that the size of data accepted by the block through this port can vary from one simulation to another without having to modify the implementation of the S-Function. This represents a great advantage if

we consider the requirement that the number of nodes, and thus the amount of data generated and processed in the model, must not be fixed.

void mdlOutputs(param)

After specifying the interface of the S-Function block to the other blocks in the model in the form of input and output ports, the behavior of the block is described in the method *mdlOutputs*. Simulink calls this method at each time step to calculate a block's outputs. In this work, the *direct feedthrough* for all the input ports is set to true, meaning that input values can be fed directly into the *mdlOutputs* method and thus used by this method to compute the output values.

Upon the implementation of the S-Function in C, the function is compiled by using the Matlab command:

mex *options* mySFunction.c

which compiles and links source files into a shared library, called a MEX-file, which is an executable from within MATLAB (e.g. a dll file in the Windows platform). In this way, the S-Function block can be executed during the simulation of the Simulink model.

For more information about S-Functions, refer to [Math05b].

7.2.3 Tool chain of DECOMSYS

The virtual prototype implemented for the validation of the AP service is based on the time-triggered paradigm. The company DECOMSYS provides customized blocks for Simulink to enable the time-triggered dispatching of tasks as well as the simulation of the time-triggered communication system FlexRay. Moreover, the customized blocks of the DECOMSYS SIMSYSTEM library represent an instrument to design the overall system architecture as a FlexRay cluster and to assign tasks (i.e. the different functions) to *hosts* (i.e. system nodes) in this cluster. In the following, the SIMSYSTEM blocks used for the design of the prototype are briefly described:

Cluster Block

The development of a FlexRay system in Simulink always begins with a Cluster block (see Figure 7-6). This block stands for the FlexRay cluster and represents the root of the system. A name for the FlexRay Cluster as well as the length of the application cycle must be provided in order for the cluster block to be correctly configured. Moreover an xcdef file, where all cluster information is saved, must be specified.

Figure 7-6 Cluster Block

Host Block

Adding a Host block to a model with a defined Cluster block is equivalent to adding a node to the modeled FlexRay cluster. Each host in the model must have a unique name and might

have one or more communication controllers. It is also possible to create a cluster with a single host, so that no communication controller is required.

Figure 7-7 Host Block

A Host block is equivalent to a subsystem in terms of Simulink. Since each host of the FlexRay cluster might execute one or more tasks, these tasks are modeled inside the host subsystem using the *Task block*.

Task Block

A Task block is a Simulink subsystem used to describe and model a task inside a host subsystem. The configuration of this block includes the specification of a period, an offset and a WCET for this task. The period of the task must be a divisor of the application cycle and means that the task is also executed cyclically. The starting time for the task within the application cycle is given by the offset. The WCET is used to check whether two tasks in the same host are conflicting.

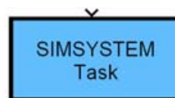

Figure 7-8 Task Block

The behavior of a task is modeled in the task subsystem by using standard or user-defined blocks. If signals are sent or received from within a task, Signal Read/Write Connectors (described in the following) must be used and a communication controller for the transmission of these signals must be specified. Here a controller is chosen from among those assigned to the host containing the task.

Dispatch Event Block

Since the Task block is a triggered subsystem, it must be connected to a Dispatch Event block (see Figure 7-9) that generates a trigger event according to the scheduling information (i.e. period, offset and WCET) specified for the task, resulting in time-triggered behavior.

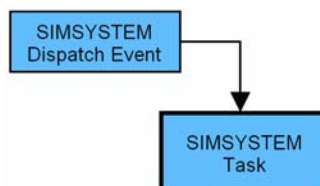

Figure 7-9 Dispatch Event Block

Signal Read/Write Connectors

Signals can be only sent and received from within a task by adding a signal read or signal write connector (see Figure 7-10) to the task subsystem. For each Signal Write Connector, several parameters can be configured, such as the name, data type or size of the signal to be

sent in bits. The only limitation here is that the size of a signal can not exceed 64 bits. Therefore, AP frames that are longer than 64 bits cannot be handled in this implementation. One or more Signal Read Connectors may receive the signal sent through a Signal Write Connector. Hence the configuration of a Signal Read Connector solely requires the specification of an existing Signal Write Connector in the model.

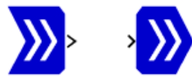

Figure 7-10 Signal Read/Write Connectors

AAFM/VP Block

By using the SIMSYSTEM blocks, an *Architecture Allocated Functional Model* is created, where not only the system functionality is described, but also the allocation of this functionality to the physical components of the system. Therefore, the model is said to be in the AAFM mode and the simulation might be started. However, the simulation in this mode only assesses the scheduling of tasks in the scope of a time-triggered operating system. In order to assess the application in combination with a FlexRay communication system, the SIMSYSTEM model must be switched to the *Virtual Prototype* mode by adding the AAFM/VP block (see Figure 7-11) and double clicking on it. The switching process, however, requires a valid communication schedule.

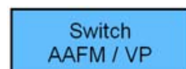

Figure 7-11 AAFM/VP Block

Upon saving the SIMSYSTEM model, all architecture details (such as hosts, communication interfaces, assignment of tasks to hosts, scheduling of tasks, etc.) and all communication-related information (e.g. signals, assignment of signals to tasks, etc.) are stored in a data repository, called *xcdef*. This file, in XML format, plays the role of interface to another tool from DECOMSYS, called DESIGNER, where the communication schedule for the FlexRay cluster can be created, based on the information available in the xcdef file and user configuration. While an automatic generation of the schedule is supported, the schedule can also be generated manually by the developer. For this, the length of the communication cycle, the length of a static slot (and thus the maximal length of a frame) and possibly the dynamic segment must be configured appropriately (see Figure 7-12).

After the successful generation of a valid communication schedule with the scheduling plug-in of DESIGNER, the communication schedule is saved to the xcdef file and therefore made available to SIMSYSTEM. It is now possible to switch the SIMSYSTEM model from the AAFM mode to the VP mode. During the switching process, the communication schedule is read from the xcdef file and all communication-relevant blocks (e.g. Signal Write/Read Connectors) are automatically transformed into SIMCOM blocks that execute according to this schedule, thus simulating the communication system FlexRay.

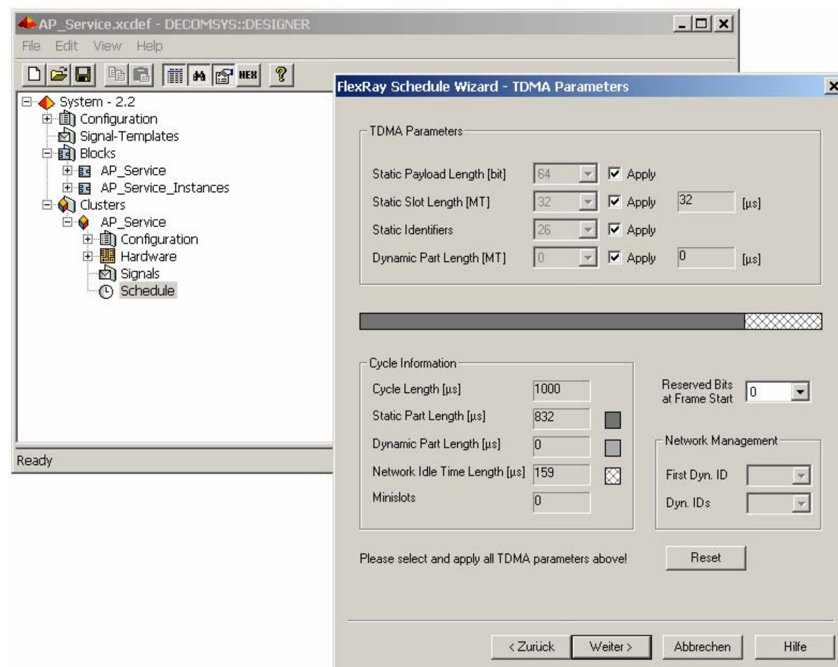

Figure 7-12 Scheduling Plug-In of DESIGNER

The modeling steps undertaken to create the virtual prototype are summarized in Figure 7-13.

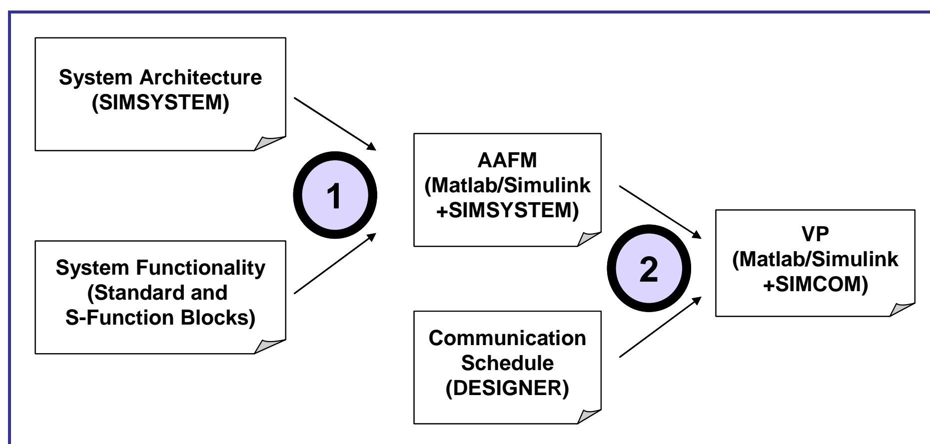

Figure 7-13 Modeling Steps

7.3 Implementation

Subsection 7.2 mainly described the different tools and steps required to create the virtual prototype. This subsection first provides a general overview on the prototype and then focuses on the implementation of the prototype's functionality. Since the AP service is a standard service, the implementation is only described for one node.

7.3.1 General Overview

The system architecture of the prototype is depicted in Figure 7-14. The system consists of *n* redundant nodes (three in the example), each reading a value from a connected sensor. Here, the sensors are emulated by Constant blocks that generate the desired value. The nodes, modeled by SIMSYSTEM Host blocks, make up a FlexRay cluster, called *AP_Cluster*, where

they communicate with each other. In order to visualize the computed results, Display blocks are connected to each Host block.

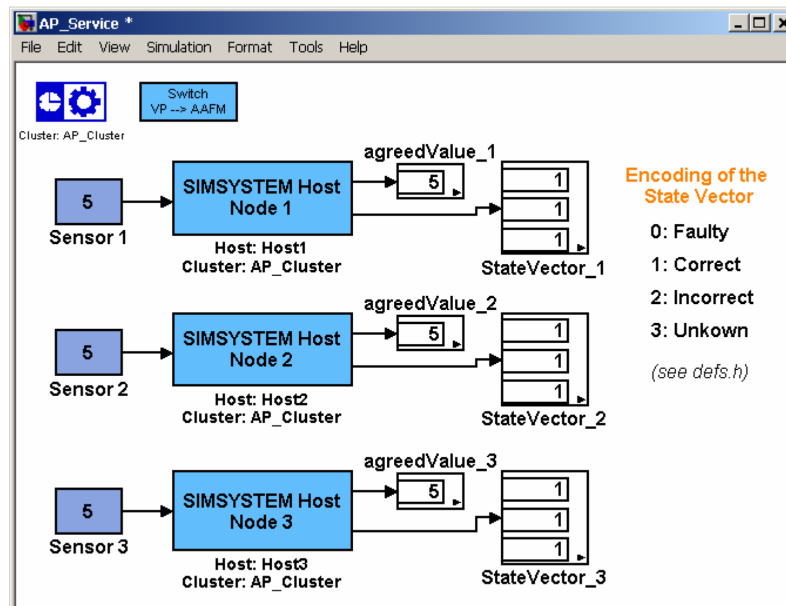

Figure 7-14 Prototype architecture

Each node emulates an application which reads the local sensor value and uses the AP service to decide, together with the other nodes, on an agreed value (see Figure 7-15).

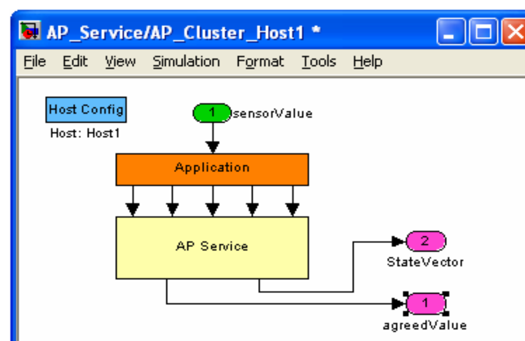

Figure 7-15 Model of a Node

In order to run the AP service, the application must deliver the input values and parameters shown in Figure 7-16. The sensor value is simply passed as the private value for the local AP service. Moreover, the node ID, a private Key for the node, the vector of public key for all the nodes (two for each node), a tolerance value and a vector of protocol parameters must be provided to the AP service. The protocol parameters are stored in a constant 1-dimensional vector, called *AP_Param* in the following order:

- 1st value: the length of the value field in an AP frame in bits.
- 2nd value: the length of the signature field in bits.
- 3rd value: the length of a node ID field in bits.
- 4th value: the total length of an AP frame in bits.

- 5th value: number of nodes involved in the agreement process.
- 6th value: number of communication phases to be executed.

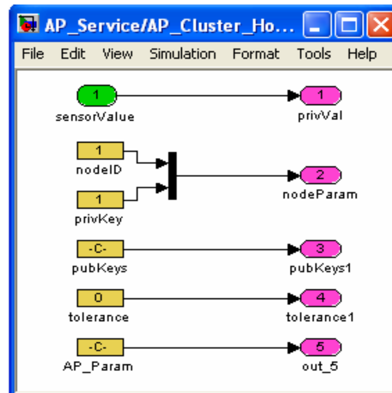

Figure 7-16 Input Values and Parameters from Application

The AP service is modeled by a set of tasks representing the protocol phases and a task for computing the service output values, as depicted in Figure 7-17. While these tasks are semantically assigned to the masking unit, their execution also requires functions from the signing unit and the decision unit. Many of these functions must handle AP frames to compute their output. Therefore the implementation of AP frames is presented in the next subsection, before a detailed description of the single units and the scheduling is given.

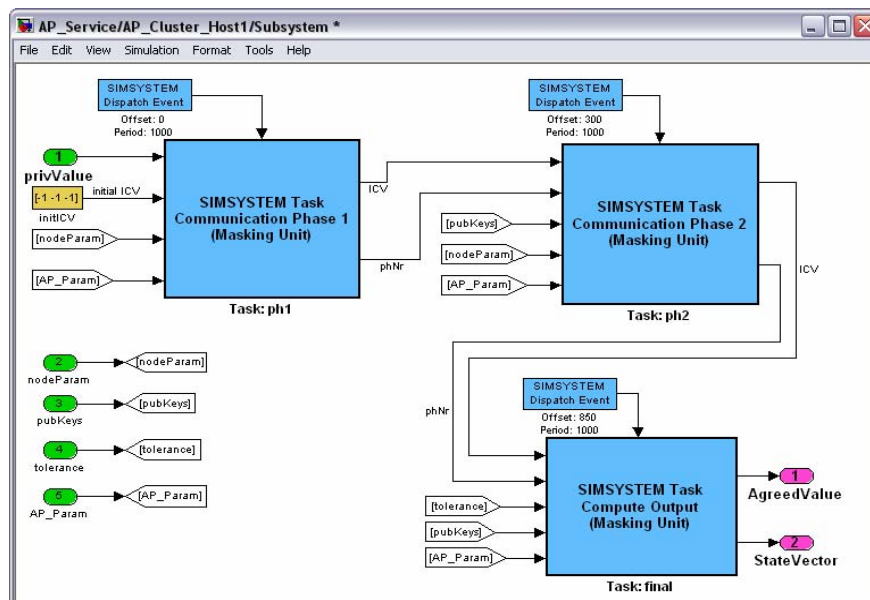

Figure 7-17 AP Service

7.3.2 Implementation of AP Frames

Many implemented functions have to use AP frames. In order to simplify the manipulation of AP frames, a record, called *AP_FRAME*, was implemented, so that the single fields of a frame can be accessed separately. This form of storage is called record format. When sent over the communication system, AP frames have the form of a 64-bit value, even though they can be shorter than 64 bits. They are then said to be in the bit format. In order to handle the two different formats, a function library (*ap_frame.h*) was implemented. This library includes the following functions:

buildFrame(param)

This function reads an *AP_FRAME* record and generates an AP frame in the bit format from the information read.

getFrameStruct(param)

This function does the opposite of the previous function. It reads an AP frame in the bit format and stores the read information in an empty *AP_FRAME* record.

getFrameValue(param)

This function reads an AP frame in the bit format and returns the value carried by the frame.

getFrameSignature(param)

This function reads an AP frame in the bit format and returns the signature field included in the frame.

getFrameNodeID(param)

This function reads out a node ID or all node IDs from an AP frame in the bit format and stores the result in a field.

7.3.3 Implementation of the Signing Unit

The signature mechanism implemented in the prototype is based on a CRC algorithm which has been implemented separately in a library file (*crc.h*). While the developer is free to choose any polynomial for the computation of the CRC, this choice only makes sense if the length of the value field and the length of the signature field specified in the AP parameters are taken into account. Since the signature results from XOR operations between the CRC of the value and one or more private keys, as described in subsection 6.3.1, its length should depend on the length of the operands. However, by specifying the exact length of the signature field in the protocol parameters, the computed signature is truncated to fit the designated field, if it is longer. In order to avoid a truncation, the keys chosen must be as long as or shorter than the specified signature length.

The signature mechanism provides the two following functionalities that are implemented by means of S-Functions:

S-Function: signFrames (sf_signFrames.c)

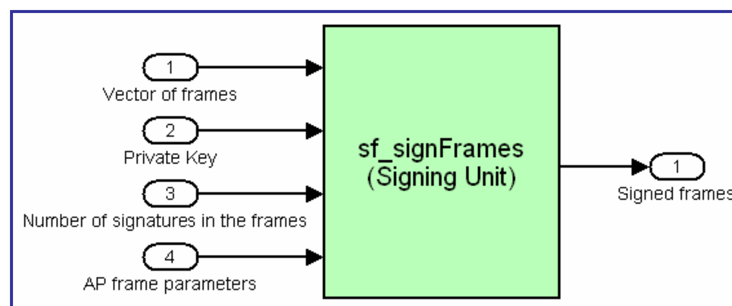

Figure 7-18 S-Function Block: signFrames

This S-function, depicted in Figure 7-18, computes the signatures for a vector of frames in the bit format by executing the following algorithm for each frame:

- Transform the frame from the input port 1 into the record format by using the function getFrameStruct from *ap_frame.h* (included in the S-Function) in order to access the value field and the signature field. For this, the frame parameters from the input port 4 are required.
- Check whether the frame has been signed using the value from the input port 3. This information can be derived from the number of the current phase where the block is inserted.
- if frame has not been signed
  - Compute the CRC for the value in the value field by applying the CRC algorithm implemented in *crc.h* (this file is included in the S-Function).
  - Compute the signature by applying (XOR operation) the private key from the input port 2 on the CRC value.
- else -- *in the case of multiple signatures*
  - Compute the new signature by applying (XOR operation) the private key from the input port 2 on the existing signature.
- Transform the updated frame record back into the bit format by using the function buildFrame from *ap_frame.h* and write the frame to the output port.

S-Function: checkSignatures (sf_checkSignatures.c)

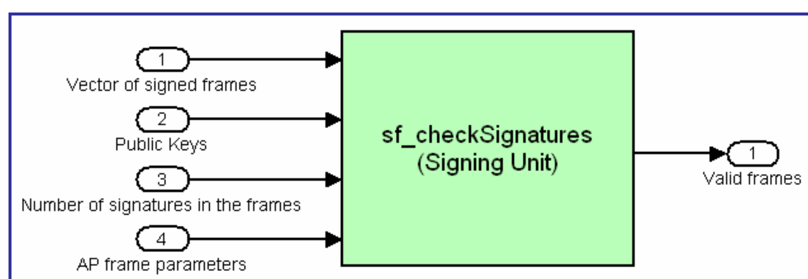

Figure 7-19 S-Function Block: checkSignatures

This S-Function, shown in Figure 7-19, checks the validity of the signatures for a vector of signed AP frames and returns only those frames which are valid. The invalid frames are replaced by empty frames (i.e. 0 values) on the output side. For each frame the following algorithm is executed:

- Transform the frame from the input port 1 into the record format by using the function getFrameStruct from *ap_frame.h* (included in the S-Function) in order to access all frame fields. For this, the frame parameters from the input port 4 are required.
- Compute the CRC for the value included in the frame by applying the CRC algorithm implemented in *crc.h* (this file is included in the S-Function).

- Identify the nodes that have signed the frame by checking the node IDs field and apply (XOR operation) the public Key 2 from the input port 2 for each of these nodes on the computed CRC, to obtain a correct extended signature *ext_signature_1*.
- Apply (XOR operation) the public Key 1 of each of the signing nodes on the signature included in the frame to obtain an extended signature *ext_signature_2*.
- if (*ext_signature_1 = ext_signature_2*) *-- i.e. the signature is valid*
    - Write the frame in the bit format to the output port.
- else
    - Write an empty frame (i.e. 0) to the output port.

This algorithm works because the extended signatures are either a combination of the CRC value, private keys and public key 1s or a combination of the CRC value and public key 2s, which are both equivalent if the signature is valid.

7.3.4 Implementation of the Decision Unit

The decision unit might implement any of the decision functions presented in subsection 0. For the prototype, a combination of the majority decision and the interval decision has been implemented as an S-Function block and called *intervalDec* (see Figure 7-20).

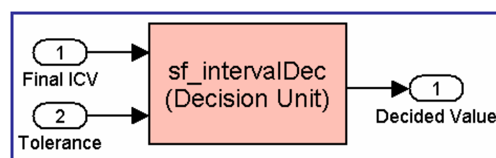

Figure 7-20 S-Function Block: intervalDec

This S-Function organizes the values in the final ICV from output port 1 in equivalence classes. In this case, two values are equivalent if their difference is smaller then the specified tolerance (input port 2). If an equivalence class containing at least the half of the values in the ICV exists, then the value corresponding to this class is written to the output port. Otherwise, *NIL* is returned, signifying that no agreement could be reached.

7.3.5 Implementation of the Masking Unit

Besides the execution of communication phases, the masking unit provides the following functionalities implemented by means of S-Functions:

S-Function: genFrame (sf_genFrame.c)

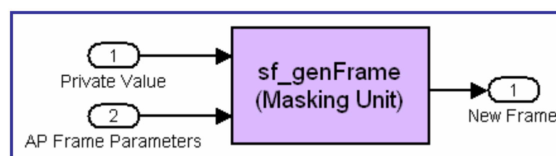

Figure 7-21 S-Function Block: genFrame

The *genFrame* S-Function, depicted in Figure 7-21, generates a new AP frame in bit format from a private value (input port 1), by using the AP frame parameters from the input port 2. The frame generated is originally unsigned and does not include the node ID of the generating

node. These fields are initially set to 0 values. It is, of course, possible to include the node ID in the frame within this S-Function. However, it is not implemented here, since this functionality is also required in other contexts and is therefore implemented in a separate S-Function (*appendID*).

S-Function: appendID (sf_appendID.c)

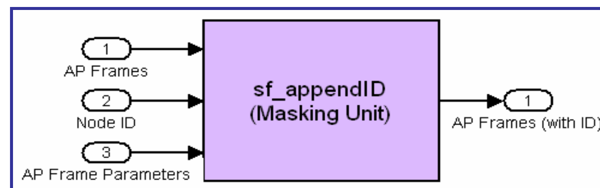

Figure 7-22 S-Function Block: appendID

This S-Function, shown in Figure 7-22, appends the node ID from the input port 2 to each of a vector of frames in the bit format from the input port 1. After transforming each frame into the record format by using the AP frame parameters from the input port 3, the new ID is appended in the final position to the vector of node IDs included in each frame. Its position is detected by the first occurrence of a 0 ID in this vector. Finally, each frame is transformed back to the bit format and written to the output port.

S-Function: updateCV (sf_updateCV.c)

The updateCV S-Function (see Figure 7-23) computes the new ICV from the AP frames received during a communication phase and the current ICV from the input port 2. Each frame from the input port 1 is transformed into record format by using the AP frame parameters from the input port 3 in order to read the value of the frame and identify the node that first propagated this value. This node is identified by reading the first ID from the node IDs vector included in the frame. The algorithm described in subsection 6.3.3 is then applied to compute the new entry in the ICV for this node.

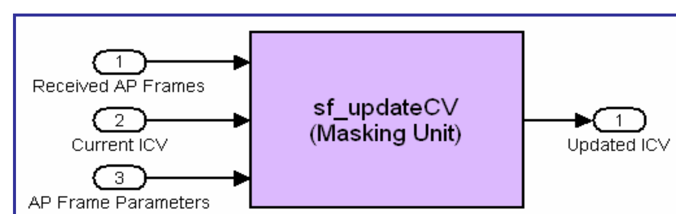

Figure 7-23 S-Function Block: updateCV

S-Function: computeSV (sf_computeSV.c)

This S-Function shown in Figure 7-24 computes the state vector from the agreed value (input port 1), the final ICV (input port 2) and the tolerance value from the input port 3 by applying the algorithm presented in subsection 6.3.3. The computed SV is then written to the output port of the block.

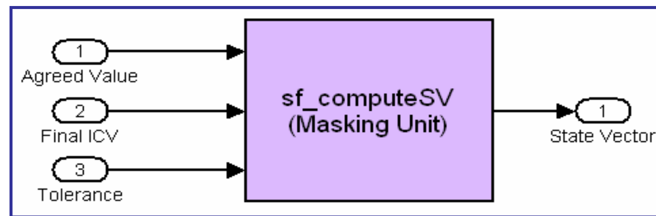

Figure 7-24 S-Function Block: computeSV

The tasks of the AP service shown in Figure 7-17 are implemented by using these functions along with the functions from the signing unit and the decision unit. Although the number of these tasks depends on the number of phases to be executed, the first task corresponding to the first communication phase and the last task, where the final output of the AP service is computed, are common to each AP service implementation.

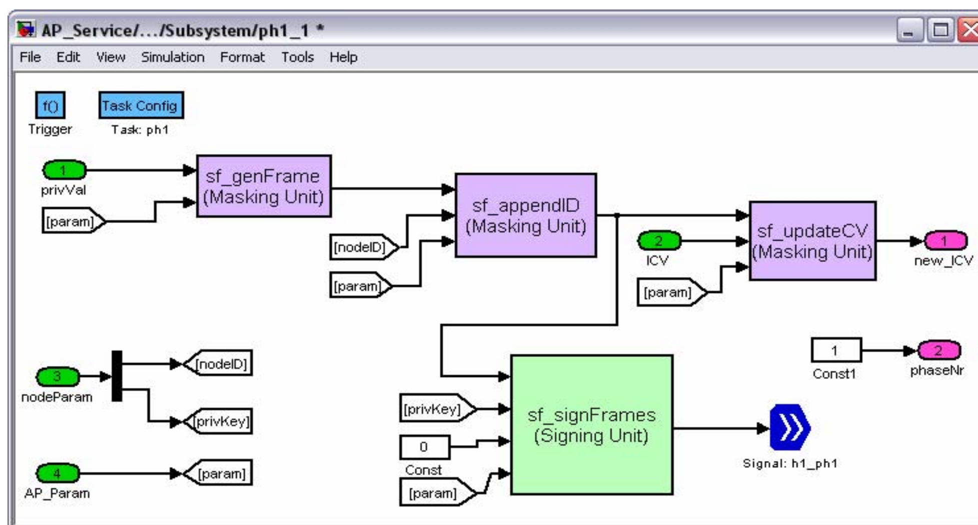

Figure 7-25 First Communication Phase

During the first phase, an AP frame is generated from the private value of the node (input port 1) by using the *genFrame* block (see Figure 7-25). After including the node ID by using the *appendID* block, the frame is signed and sent over the simulated communication system FlexRay as a 64-bit signal by using a Signal Write Connector. The ICV input to this phase (input port 2) consists originally of *NIL* values for each node. This ICV is updated with the private value of the node before being passed onto the subsequent phase (output port 1) along with the current phase number (output port 2) which is equal to 1 for the first phase. The phase number is used by the next phase to identify the number of signatures in the frames received in that phase.

Depending on the number of the phases to be executed, one or more tasks for communication phases are included in the AP service.

These communication phases are modeled as shown in Figure 7-26. At the beginning of the phase, all AP frames sent to the node are received by means of Signal Read Connectors. These frames are merged into a vector by using a Mux block and fed to a *checkSignatures* block where invalid frames are detected. The vector of valid frames is input to the *updateCV* block, where the new ICV is computed (output port 1). Additionally, the executing node appends its ID (*appendID* block) to each of the frames in the vector, signs it (*signFrames*

block) and sends it over the communication system using a Signal Write Connector, after splitting the vector into single frames with a Demux block. The number of the current phase is finally computed and passed to the next phase (output port 2).

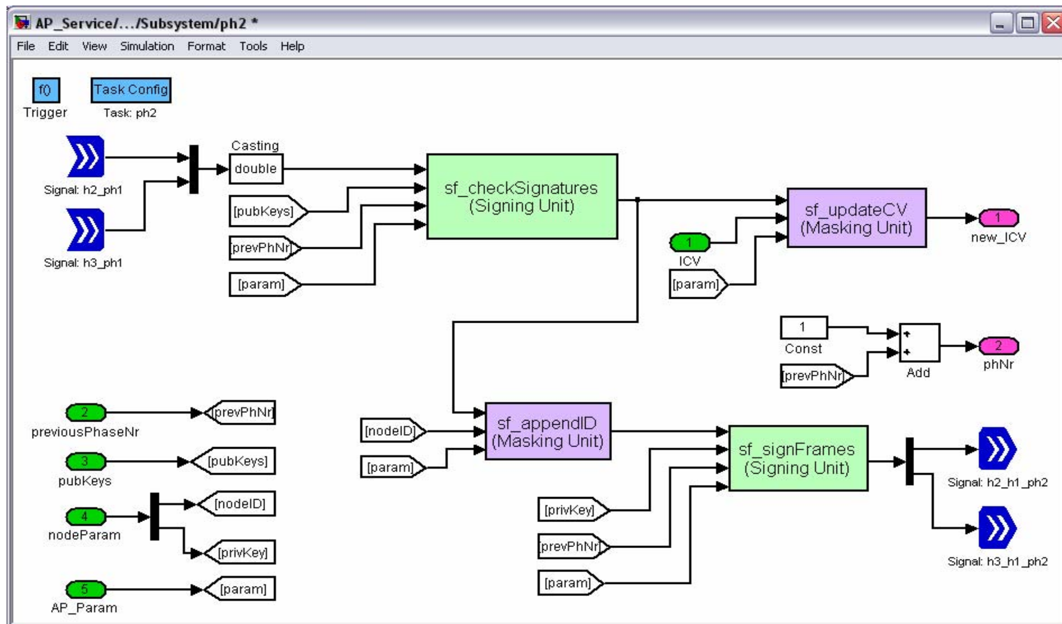

Figure 7-26 Standard Communication Phase

In order to add a new communication phase, the developer only needs to add a copy of the second task (Communication Phase 2, see Figure 7-17) to the AP service subsystem (see Figure 7-15). Since an additional communication phase implies an additional exchange of AP frames, the Signal Read/Write blocks in all tasks must also be configured appropriately.

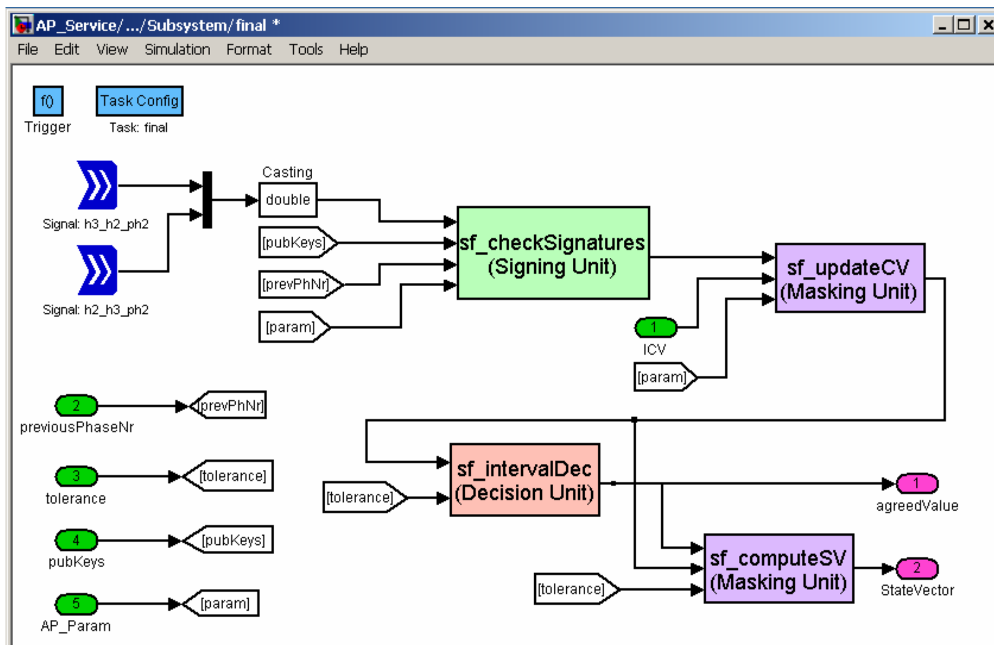

Figure 7-27 Final Task

Within the final task, shown in Figure 7-27, the AP frames sent in the last communication phase are received. After a signature check, the valid frames are fed to the *updateCV* block to compute the final ICV. This ICV is then input into the *intervalDec* block which delivers an

agreed value (output port 1). These values along with the tolerance value (input port 3) are finally used by the *computeSV* block to compute the SV (output port 2).

7.3.6 Scheduling of SIMSYSTEM Tasks

Although the tasks of the AP service are executed in sequence within a node, similar tasks must be executed simultaneously in all nodes. Therefore, a uniform scheduling of tasks is required. Since the AP service executes cyclically, the schedule describes only one execution round of all the tasks, called an application cycle.

Each task is scheduled separately by configuring the corresponding Task block. For a reasonable overall scheduling of the tasks, it is important to take the execution delays as well as the communication delays into account, so that enough time between tasks is scheduled for both the execution of single tasks and the transmission of AP frames. It is also important to keep in mind that FlexRay frames can not be transmitted simultaneously due to the TDMA mechanism, as opposed to the execution of similar tasks. This means that the overall communication delay between two successive communication phases increases proportionally with the number of nodes.

For the prototype, the schedule shown in Figure 7-28 is suggested. This schedule is only an example and must be adapted to the application cycle of the real application making use of the AP service, since the private values are also provided on a cyclic basis from the application.

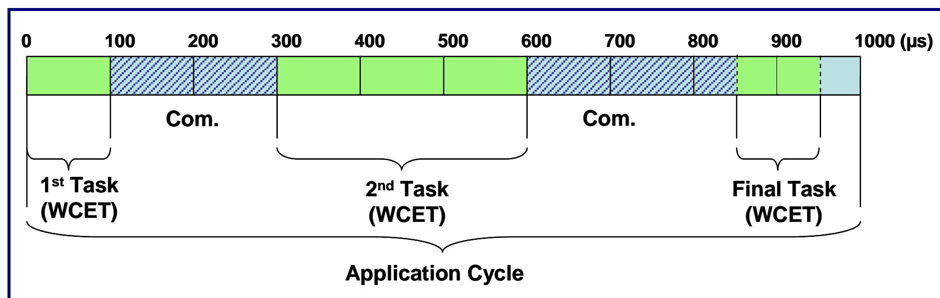

Figure 7-28 Scheduling of Tasks in a Node

7.3.7 Generation of the Communication Schedule

The communication schedule is generated by the scheduling plug-in of DESIGNER. This plug-in does not only take transmission delays into consideration, but also the delay of so called LLIO tasks (Low Level In Out). These tasks are created automatically during the transformation of the SIMSYSTEM model into a virtual prototype. They simulate the interactions between the application level and the communication controller, i.e. the writing of signals to the buffers of the communication controller, where the actual FlexRay frame is generated, and the reading of received signals from these buffers. Receiving LLIO tasks are scheduled right before the application task receiving the signal, while sending LLIO tasks are scheduled after the sending application task, as shown in Figure 7-29. The estimated delay for these tasks can be configured in DESIGNER before the communication schedule is generated.

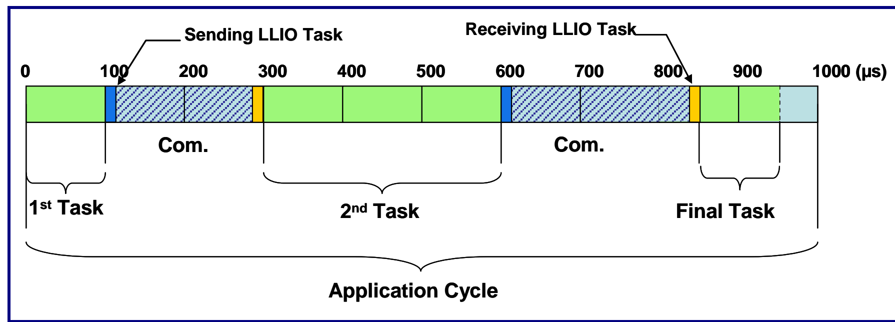

Figure 7-29 Scheduling of LLIO Tasks

Besides information regarding the system architecture, the scheduling plug-in also receives information concerning the scheduling of tasks and assignment of communication signals to the single tasks from the xcdef file. This information is also taken into consideration during the generation of a communication schedule in order to guarantee that signals are received on time by the corresponding task.

The FlexRay communication schedule is executed in a cyclic manner, as described in subsection 4.3.3. The cycle length can be configured by the developer and must be a divisor of the application cycle which is read from the xcdef file. This makes sense, since signals are sent and received by tasks whose execution cycle is also a divisor of the application cycle. It also ensures that the task dispatching schedule and the communication schedule can be executed synchronously.

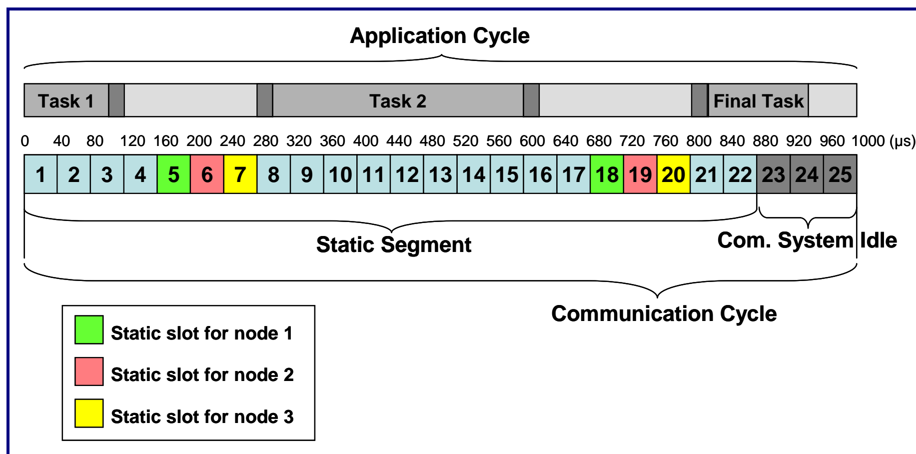

Figure 7-30 Communication Schedule

For the virtual prototype consisting of three nodes, each node sends one 64-bit signal during the first communication phase and two signals during the second communication phase. By choosing the payload of a FlexRay frame to be at least 128-bit long, the two signals of the second phase can be combined into a single frame, so that each node sends a single frame and thus requires a single communication slot in each communication phase. The length of the communication cycle is chosen to be equal to the length of the application cycle and the length of the dynamic segment is set to 0. The resulting communication schedule is shown in Figure 7-30 in combination with the application schedule. During each phase, every node is assigned a communication slot in the static segment to transmit its FlexRay frame before the next phase begins. As the figure also shows, the sending LLIO tasks must be scheduled before the transmission of the FlexRay frames begins, while receiving LLIO tasks must be scheduled

after the transmission of frames is finished, to ensure that the data in the communication controller buffers is not outdated.

8 Evaluation

In order to evaluate the AP service, different configurations must be tested using the implemented virtual prototype, so that the behavior of the prototype can be assessed under different circumstances. These configurations, which correspond to evaluation cases, are represented by different sets of input data. In some of these evaluation cases, the prototype must be modified in order to enforce a particular operation state. Since applications in the automotive domain will involve at most three redundant nodes (i.e. a triplex system), most of the evaluation cases (i.e. case 1 to case 7), presented here, consider this configuration. Moreover, the number of nodes assumed to be faulty does not exceed one node. In two evaluation cases (i.e. case 6 and case 7), the behavior of the AP service in the presence of communication faults is assessed. An additional test case (i.e. case 8) will demonstrate the scalability of the AP service within a four-node system configuration. Excluding the specified set of input data, no initial conditions must be fulfilled. The input data for each node is specified by a pair in the form (private value, tolerance), while the output consists of an agreed value and a state vector, where the states are encoded as follows:

0: FAULTY

1: CORRECT

2: INCORRECT

3: UNKNOWN

In the following subsections, the evaluations cases simulated in the virtual prototype are described.

8.1 Evaluation Case 1

The first evaluation case should prove the functionality of the implemented concept. The agreement protocol here involves three nodes having the same private value 5 and a tolerance value equal to 0. The output expected is depicted in Table 8-1.

Input	Expected Output
Node 1: (5,0) Node 2: (5,0) Node 3: (5,0)	Node 1: (5, (1, 1, 1)) Node 2: (5, (1, 1, 1)) Node 3: (5, (1, 1, 1))

Table 8-1 Evaluation Case 1

8.1.1 Evaluation Procedure and Results

In order to execute this evaluation case, the virtual prototype is initiated in Simulink. Then, Constant blocks corresponding to the three different sensors are configured to deliver the values shown in Table 8-1. The tolerance value for each node is set to 0 by configuring the corresponding Constant block in the Application subsystem of each node to output the value

0. Finally, the Simulink model is executed and the results are visualized in Display blocks, as shown in Figure 8-1.

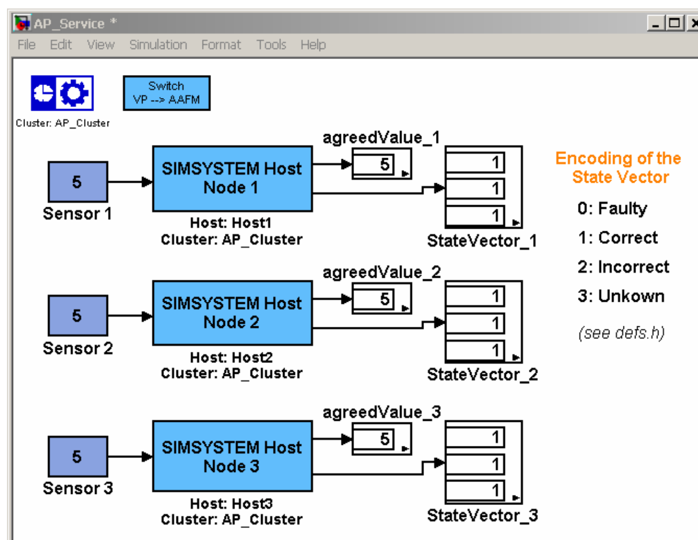

Figure 8-1 Prototype Execution for Case 1

8.1.2 Evaluation of the Results

The execution of the prototype in this configuration delivers the expected results, indicating that the implemented AP service is functional.

8.2 Evaluation Case 2

In this evaluation case, the behavior of the protocol in the presence of a faulty node is tested in order to prove fault-tolerance and the ability to detect faulty nodes. Here, node 1, marked by a star in Table 8-2, sends inconsistent values to nodes 2 and 3 and is therefore faulty.

Input	Expected Output
Node 1*: (5,0)	Node 1*: (5, (1, 1, 1))
Node 2: (5,0)	Node 2: (5, (0, 1, 1))
Node 3: (5,0)	Node 3: (5, (0, 1, 1))

Table 8-2 Evaluation Case 2: One Faulty Node

8.2.1 Evaluation Procedure and Results

For this case we assume that each node replicates its private value before creating two separate AP frames that are sent separately to the other nodes during the first communication phase. The execution of this case therefore requires a modification of the prototype to simulate this behavior, as well as a replication fault in node 1, so that an AP frame containing the original private value is sent to node 2, while an AP frame containing the value 7 is sent to node 3, as shown in Figure 8-2.

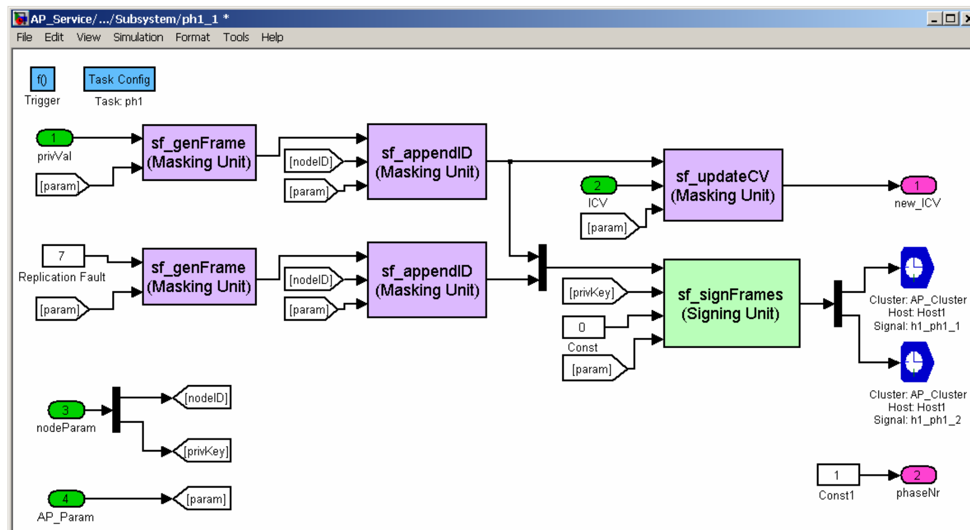

Figure 8-2 Emulation of a Replication Fault in Node 1

The private values as well as the tolerance value must be configured appropriately, as described in subsection 8.1.1. The execution of the prototype in this configuration results in the output shown in Figure 8-3.

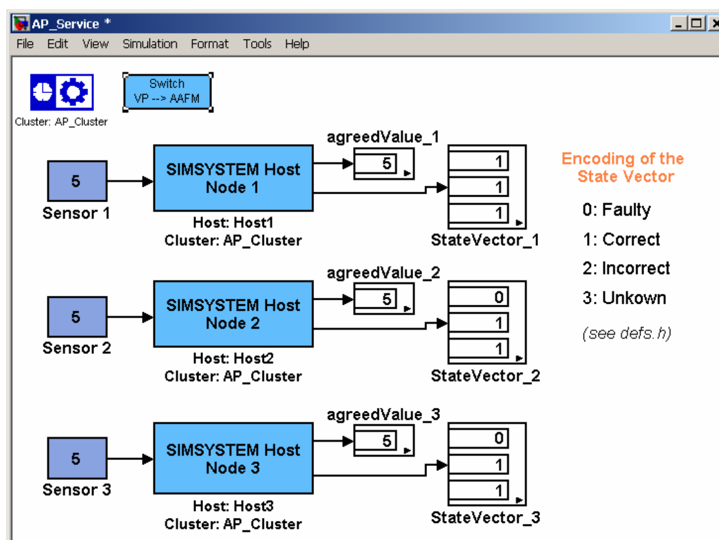

Figure 8-3 Prototype Execution for Case 2

8.2.2 Evaluation of the Results

The simulation of this evaluation case delivers the expected results. Nodes 2 and 3 reached agreement despite the faulty node 1 and also identified this node as being faulty. Therefore, both fault-tolerance and the capability for fault-detection in the concept implemented, have been validated.

8.3 Evaluation Case 3

This evaluation case aims to validate the behavior of the AP service in the presence of non-determinism. Moreover, no deviations are tolerated by the application, i.e. tolerance = 0. The expected output is shown in Table 8-3.

Input	Expected Output
Node 1: (5,0)	Node 1: (5, (1, 1, 2))
Node 2: (5,0)	Node 2: (5, (1, 1, 2))
Node 3: (4,0)	Node 3: (5, (1, 1, 2))

Table 8-3 Evaluation Case 3: Non-Determinism, Tolerance = 0

8.3.1 Evaluation Procedure and Results

The private values as well as the tolerance value are configured as described in subsection 8.1.1. The execution of the prototype in this configuration results in the output shown in Figure 8-4.

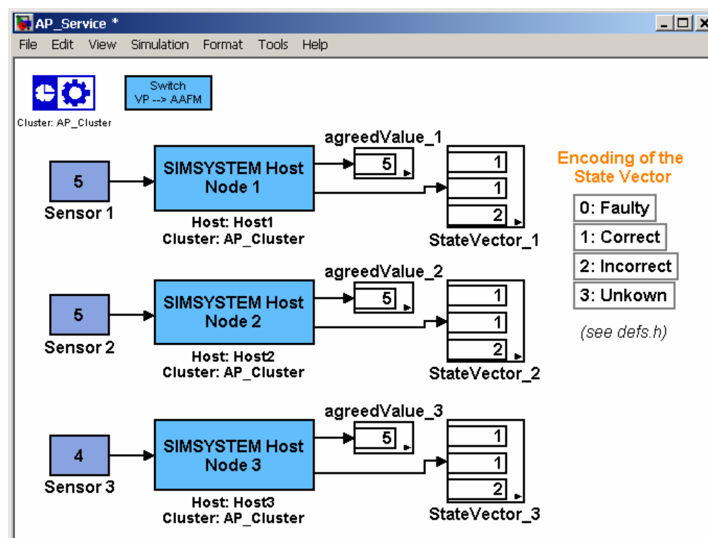

Figure 8-4 Prototype Execution for Case 3

8.3.2 Evaluation of the Results

The simulation of this evaluation case delivers the expected results. The three nodes reached agreement despite non-determinism and detected that the private value of node 3 is incorrect.

8.4 Evaluation Case 4

This evaluation case is designed to test the behavior of the AP service in the presence of non-determinism when deviations smaller or equal to 1 are tolerated. The input and the expected output are depicted in Table 8-4.

Input	Expected Output
Node 1: (5,1)	Node 1: (5, (1, 1, 2))
Node 2: (4,1)	Node 2: (5, (1, 1, 2))
Node 3: (7,1)	Node 3: (5, (1, 1, 2))

Table 8-4 Evaluation Case 4: Non-Determinism, Tolerance = 1

8.4.1 Evaluation Procedure and Results

The private values as well as the tolerance value are configured as described in subsection 8.1.1. Figure 8-5 shows the outcome of the prototype execution in this configuration.

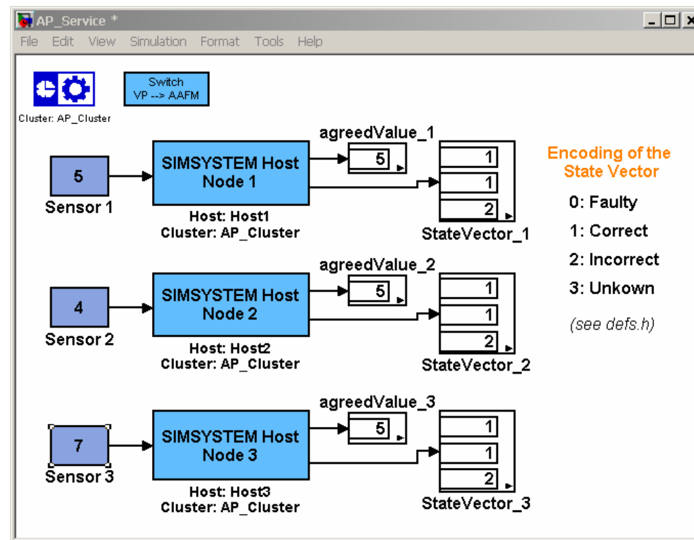

Figure 8-5 Prototype Execution for Case 4

8.4.2 Evaluation of the Results

The execution results are similar to the results expected. Deviations that are smaller than the specified tolerance value 1 are accepted (e.g. value of sensor 2 in Figure 8-5). Deviations that are greater than 1 are not tolerated (e.g. value of sensor 3 in Figure 8-5). Moreover, all nodes agree upon the value 5, since they are non-faulty.

8.5 Evaluation Case 5

This evaluation case is similar to the evaluation case 2 concerning the faulty behavior of node *A*. However, it also includes non-determinism, so that the private value of node *C* differs from the other values. Moreover, the application tolerance is set to 1. This case aims to validate the behavior of the AP service when non-determinism is combined with the presence of a faulty node. The results expected for this configuration are shown in Table 8-5.

Input	Expected Output
Node 1*: (5,1)	Node 1*: (5, (1, 1, 1))
Node 2: (5,1)	Node 2: (5, (0, 1, 1))
Node 3: (4,1)	Node 3: (5, (0, 1, 1))

Table 8-5 Evaluation Case 5: 1 Faulty Node, Non-Determinism, Tolerance = 1

8.5.1 Evaluation Procedure and Results

The execution of this evaluation case requires the modification of the virtual prototype as in subsection 8.2.1. Moreover, the sensor value and the tolerance value for each node have to be

configured appropriately. The execution of the prototype for this configuration delivers the output depicted in Figure 8-6.

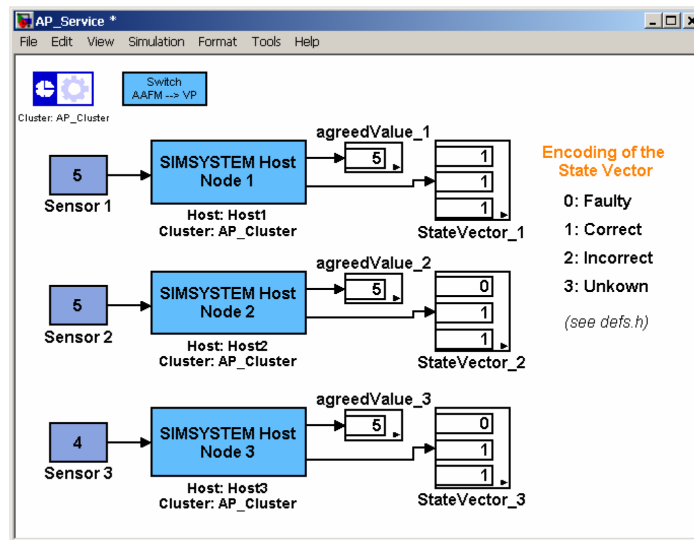

Figure 8-6 Prototype Execution for Case 5

8.5.2 Evaluation of the Results

The execution output for this evaluation case is as expected. The non-faulty nodes 2 and 3 reached agreement despite non-determinism and in presence of a faulty node (i.e. node 1), which they identified as such. Hence, it has been demonstrated that the combination of these two factors does not affect the functionality of the AP service.

8.6 Evaluation Case 6

This evaluation case demonstrates the behavior of the AP service, when a transient communication fault occurs during the first communication phase leading to the loss of all frames from a particular node. For this case, we assume that the frames from node 1 are lost. The virtual prototype configuration and the expected execution output are depicted in Table 8-6.

Input	Expected output
Node 1: (4,1) Node 2: (5,1) Nod3 3: (5,1)	Node 1: (4, (1, 1, 1)) Node 2: (5, (3, 1, 1)) Node 3: (5, (3, 1, 1))

Table 8-6 Evaluation Case 6: Communication Fault (1st Com. Phase)

8.6.1 Evaluation Procedure and Results

The execution of this evaluation case requires the modification of the virtual prototype in order to emulate a communication fault. Therefore, the Signal Write Connector in the first task (i.e. first phase) of the AP service in node 1 is removed. The corresponding Signal Read Connectors in the second task in node 2 and node 3 are replaced by Constant blocks that

output the value 0 which corresponds to a default frame value. Moreover, the sensor values and the tolerance value for each node must be properly configured. The execution output is shown in Figure 8-7.

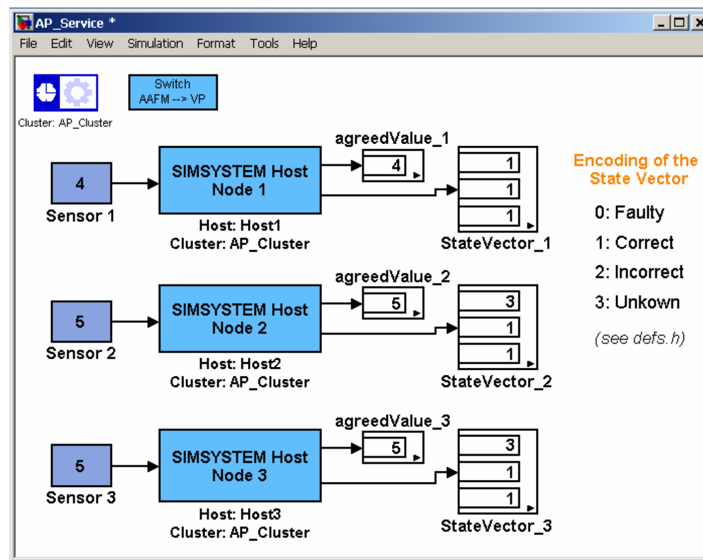

Figure 8-7 Prototype Execution for Case 6

8.6.2 Evaluation of the Results

As discussed in subsection 6.3.4, a communication fault during the first communication phase can lead to disagreement among non-faulty nodes. However the agreement of the nodes 2 and 3 on the state *UNKNOWN* for node 1 indicates that frames expected from this node have not been correctly received.

8.7 Evaluation Case 7

This evaluation case is similar to the previous one with the difference that the communication fault occurs during the second communication phase leading to the loss of all AP frames forwarded by node 1 during this phase. Once again, the behavior of the AP service is to be assessed based on the discussion about communication faults in subsection 6.3.4. The input values, as well as the expected output, are depicted in Table 8-8. The agreed value must be the value 4 because this value occurs first in every ICV and all the other values (i.e. 5) are in the same interval of width 1 (i.e. the tolerance) as this value.

Input	Expected output
Node 1: (4,1) Node 2: (5,1) Node 3: (5,1)	Node 1: (4, (1, 1, 1)) Node 2: (4, (1, 1, 1)) Node 3: (4, (1, 1, 1))

Table 8-7 Evaluation Case 7: Communication Fault (2nd Com. Phase)

8.7.1 Evaluation Procedure and Results

The execution of this evaluation case requires the modification of the virtual prototype in order to emulate a communication fault causing the loss of all frames sent by node 1 during the second communication phase. Therefore, the Signal Write Connectors in the second task (i.e. second phase) of the AP service in node 1 are removed. The corresponding Signal Read Connectors in the final task in node 2 and node 3 are replaced by Constant blocks that output the value 0, which corresponds to a default frame value. Moreover, the sensor values and the tolerance value for each node must be properly configured. The execution output is shown in Figure 8-8.

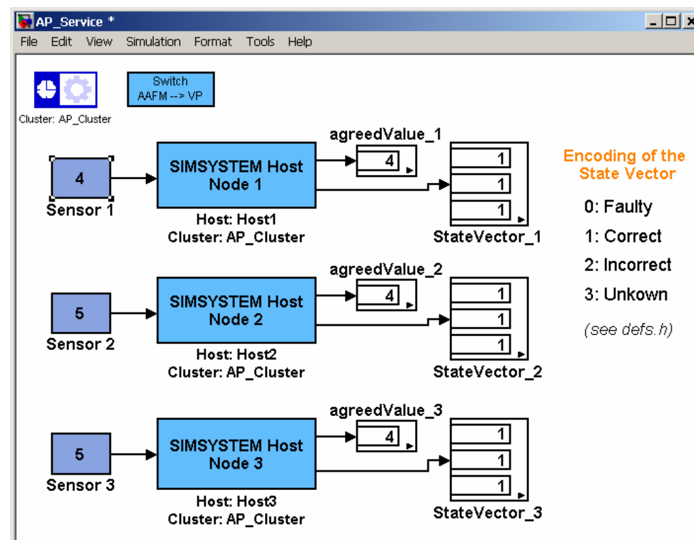

Figure 8-8 Prototype Execution for Case 7

8.7.2 Evaluation of the Results

The execution of the virtual prototype in this configuration delivered the expected output, which confirms that communication faults occurring during the second phase can be always tolerated, as long as they are not combined with a node fault.

8.8 Evaluation Case 8

In this evaluation case, the scalability of the AP service is to be proved. Table 8-8 shows the prototype configuration for this case, as well as the expected results.

Input	Expected Output
Node 1: (5,0)	Node 1: (5, (1, 1, 1, 1))
Node 2: (5,0)	Node 2: (5, (1, 1, 1, 1))
Node 3: (5,0)	Node 3: (5, (1, 1, 1, 1))
Node 4: (5,0)	Node 4: ((5, (1, 1, 1, 1))

Table 8-8 Evaluation Case 8: 4-Node Configuration

8.8.1 Evaluation Procedure and Results

The execution of this test requires the expansion of the virtual prototype by a 4th node which is fundamentally a copy of one of the existing nodes. The configuration for this case includes assigning an additional ID and the required keys (a copy of another node's key set should be sufficient) to the new node, changing the initial ICV for all the nodes to a 4-value vector (i.e. (*NIL*, *NIL*, *NIL*, *NIL*)), setting the number of nodes in the AP service parameters (*AP_Param*) for each node to 4 and finally adding the additional signals to both communication phases, i.e. the AP frames generated in node 4 and those forwarded by this node. The execution results of the prototype for this configuration are shown in Figure 8-9.

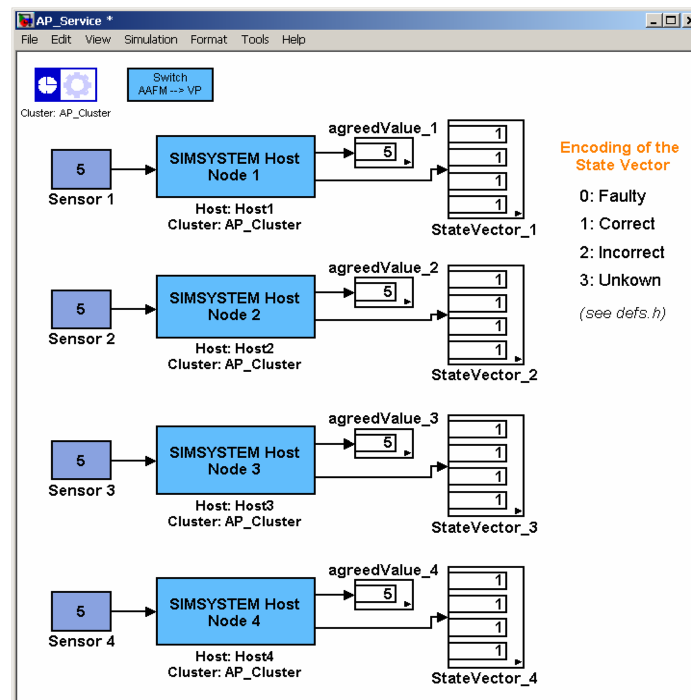

Figure 8-9 Prototype Execution for Case 8

8.8.2 Evaluation of the Results

The virtual prototype delivered the expected output for this evaluation case, which validates the functionality of the AP service for systems larger than the triplex configuration.

8.9 Conclusion

The simulation of the presented cases aimed to assess the Agreement Protocol service concept suggested in section 6 with respect to its functionality, its fault-tolerance, its behavior in the presence of communication failures and its scalability. Since the output for each case matched the results expected, all these aspects have been validated.

9 Summary and Outlook

9.1 Summary

The amount of safety-relevant electronics in modern vehicles is growing rapidly due to the increasing customer requirements for passenger safety. Since the majority of the functions in electronic systems are realized in software, for reasons of costs and flexibility, the amount of embedded software has been also growing at a rapid pace, combined with an increasing complexity and heterogeneity. In order to master this complexity, standard software architectures for automotive embedded software have been developed, providing developers with the required basic services, so that they can concentrate their efforts on application specific software. Consequently, the developed software is mainly platform-independent, has a higher quality and the development costs are lower, since proven basic software components are reused. In the scope of the EU-project EASIS, the focus lies in integrated safety applications. Therefore dependability services are being developed, so that they can be used by these applications as standard software modules. In this diploma thesis (Diplomarbeit), an Agreement Protocol service has been conceived and developed in Matlab/Simulink as a standard dependability software module to be integrated in the EASIS software platform.

Agreement protocols are used as an instrument of fault-tolerance to enable redundant system components to reach agreement upon a particular dimension (e.g. velocity, steering angle, system state, etc.) in a fault-tolerant manner. In this way, it can be guaranteed that the subsequent behavior of these components is uniform. Several approaches to the implementation of an agreement protocol as specified in the literature have been analyzed and compared. The Signed Messages protocol was most suitable for an implementation as a standard service in the scope of EASIS. However, changes had to be made to the original algorithm, as the requirements and the specific constraints of EASIS had to be taken into consideration. The integration of the AP service in the EASIS software architecture involved the identification of other services from EASIS that might be used, such as the FT-Com service, and the specification of the service interface.

The AP service has been designed as a set of three units. The masking unit represents the core of the service as it is responsible for the execution of the agreement protocol and the computation of the output values. Since the signature mechanism and the decision mechanism depend on the particular application using the AP service, they have been integrated in separate, exchangeable units: the signing unit and the decision unit. This design allows for flexibility in the implementation of the AP service with different applications.

Since the values exchanged between the nodes within the scope of the agreement protocol must be signed, a concept for an AP frame has been suggested, so that the value, the signature information and other protocol information can be packed together in a structured manner into a single message.

As opposed to IT applications, automotive applications do not require a cryptographically strong signature mechanism. Therefore a simple and efficient signature mechanism, which is

based on CRC computations and XOR operations, has been developed for use within the scope of the AP service.

In order to evaluate the concepts suggested, a virtual prototype has been implemented in Matlab/Simulink. Third party software tools have been used to simulate a time-triggered operating system and the time-triggered communication system FlexRay in the scope of this prototype. For the validation, several evaluation cases have been designed and simulated in the virtual prototype, taking into account the relevant use cases. The evaluation results have been very satisfactory and hence validated the implemented concept.

9.2 Outlook

In order to take early validation one step further with regard to the accuracy and reliability of test results, rapid prototyping, an approach consisting of integrating the virtual prototype into a real time environment, is practiced.

Figure 9-1 MicroAutoBox from dSpace [Dspa05]

Therefore, the hardware platform MicroAutoBox from the company dSpace is used (see Figure 9-1). This represents a compact, stand-alone prototyping-hardware with real-time support, I/O and signal conditioning, used for the rapid and reliable test of control functions in real-time. In order to compile the virtual prototype for this hardware target, the Simulink model has to be expanded by specific Real-Time-Interface blocks provided by dSpace [Dspa05]. The machine code can then be generated automatically for each system node by using Real-Time-Workshop. Moreover, the MicroAutoBox units include a FlexRay communication controller and can, therefore, be connected by a FlexRay bus that ensures communication between the system nodes.

Currently, two different functions are under discussion for the validation of the agreement protocol within a SafeLane application:

- **Agreement upon input sensor values:** redundant sensor information must undergo an agreement protocol before being processed.
- **Agreement upon application status:** enables different components of a system to agree upon the status of a distributed application (i.e. the SafeLane application).

Depending on the decisions that are made concerning the validation of the EASIS services in the scope of the EASIS project, one of these functions will be implemented using the Agreement Protocol service.

Literature

[Benz04] **Benz S.**: *Eine Entwicklungsmethodik für sicherheitsrelevante Elektroniksysteme im Automobil.* University of Karlsruhe, Germany, 2004

[BuWe01] **Burns A.; Wellings A.**: *Real-Time Systems Programming and Programming Languages*, 3rd edition, Harlow, Munich: Addison-Wesley, 2001

[Deco05] http://www.decomsys.com, 2005

[Dspa05] http://www.dspace.de, 2005

[Easi04a] **The EASIS Consortium**: *Deliverable D0.2.4 – EASIS General Architecture Framework*. At: http://www.easis-online.org, 2004

[Easi05a] DaimlerChrysler internal EASIS Documentation, 2005

[Easi05b] http://www.offis.de/projekte/EASIS, 2005

[Easi05c] http://www.easis-online.org, 2005

[East05] http://www.east-eea.net, 2005

[EcBe02] **Echtle K.; Best A.**: *Standardverfahren und spezielle Ansätze zur Fehlertoleranz von Steuergeräten in Automobilen.* DC internal Documentation, 2002

[Echt90] **Echtle K.:** *Fehlertoleranzverfahren.* Studienreihe Informatik. Berlin, Heidelberg: Springer-Verlag, 1990

[Flex02] DaimlerChrysler internal FlexRay Documentation, 2002

[Hede01] **Hedenetz B.**: *Entwurf von verteilten fehlertoleranten Elektronikarchitekturen in Kraftfahrzeugen.* Dissertation, Tübingen, Germany, 2001

[Hein04] **Heinecke H.**: *An industry-wide initiative to manage the complexity of emerging Automotive E/E-Architectures.* In: Convergence International Congress & Exposition On Transportation Electronics, Detroit, MI, USA, Session: Systems Architecture: Software, 2004

[Hell05] **Hellestrand G.**: *ESL Development Gets A Leg Up*. In: Chip Design Magazine, December / January 2005

[Hell04] **Hellestrand G.**: *Model based development with virtual prototypes.* Technical paper from the company VaST System Technology, 2004

[HRK+00] **Hedenetz B.; Ruh J.; Kühlewein M. et al.**: *OSEKtime – the new fifth OSEK/VDX group: A Dependable Fault-Tolerant Real-Time Operating System and Communication Layer for By-Wire Applications.* VDI-

Verlag, 2000

[HSB+02] **Heinecke H.; Schedl A.; Berwanger J. et al.**: *FlexRay – ein Kommunikationssystem für das Automobil der Zukunft*, In: www.elektroniknet.de, 2002

[IrJu04] **Irion J.; Junker M.**: *Embedded System Technology Platform*. In: ARTEMIS European Technology Platform on Embedded Systems, Rome, 2004

[LSP80] **Lamport L.; Shostak R.; Pease M.**: *Reaching Agreement in the Presence of Faults*. In: Journal of the ACM, vol. 27, no. 2, pp. 228..234, April 1980

[LSP82] **Lamport L.; Shostak R.; Pease M.**: *The Byzantine Generals Problem*, In: ACM Transactions on Programming Languages and Systems, vol. 4, no. 3, pp. 382..401, July 1982

[Math05a] http://www.mathworks.com, 2005

[Math05b] http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/sfg, 2005

[Osek05] http://www.osek-vdx.org, 2005

[Ring01] **Ringler T. K.**: *Entwicklung und Analyse zeitgesteuerter Systeme*. Dissertation , Stuttgart, Germany, 2001

[Seth04] **Sethna F.**: *CAN and FlexRay for Vehicle Communications*. University of Sussex, England, 2004

[Vect05] http://www.vector-informatik.com/english, 2005

[Voge03] **Voget S.**: *Future trends in Software Architectures for Automotive Systems*. In: AMAA yearbook, 2003

[XuPa93] **Xu J., Parnas D.L.**: *On satisfying timing constraints in hard-real-time systems*. In: IEEE Transactions on Software Engineering, vol. 19, no. 1, pp. 70..84, January 1993

APPENDIX A – Communication Overhead of the OM and SM Algorithms

Derivation of the minimal and maximal communication overhead for the OM algorithm:

$$\underbrace{n\cdot(n-1)}_{\text{1st Round}}+\underbrace{n\cdot(n-1)^2}_{\text{2nd Round}}+\ldots+\underbrace{n.(n-1)^{m+1}}_{\text{(m+1)-th Round}}=$$

$$n\cdot((n-1)+(n-1)^2+\ldots+(n-1)^{m+1})=$$

$$\frac{n\cdot(n-1)((n-1)^{m+1}-1)}{n-2}$$

Derivation of the minimal and maximal communication overhead for the SM algorithm:

$$\underbrace{n\cdot(n-1)}_{\text{1st Round}}+\underbrace{n\cdot(n-1)\cdot(n-2)}_{\text{2nd Round}}+\ldots+\underbrace{n.(n-1)\cdot\ldots\cdot(n-m-1)}_{\text{(m+1)-th Round}}=$$

$$\frac{n!}{(n-2)!}+\frac{n!}{(n-3)!}+\ldots+\frac{n!}{(n-m-2)!}$$

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Unterschrift:

Stuttgart, den 22.11.2005