DISSERTATION

Diagnosis and Maintenance in an Integrated Time-Triggered Architecture

ausgeführt zum Zwecke der Erlangung des
akademischen Grades eines

Doktors der technischen Wissenschaften

unter der Leitung von

O. Univ.-Prof. Dr. Hermann Kopetz
Institut für Technische Informatik 182

eingereicht an der

Technischen Universität Wien
Fakultät für Informatik

von

Philipp Peti
Matr.-Nr. 9625740
Rittergasse 1/2/10, A-1040 Wien

Wien, im September 2005 .....

Diagnosis and Maintenance in an Integrated Time-Triggered Architecture

The use of electronics in automotive and avionic industry accounts for numerous improvements with respect to system safety, fuel economy, passenger comfort, and system costs. In last years, however, the development of effective diagnostic systems stayed behind the recent increase of electronic systems in modern means of transportation. In combination with shrinking geometries this issue has lead to a dramatic increase of transient system failures that cannot be traced. This so called trouble-not-identified phenomenon is a significant problem with major economic implications due to false maintenance actions resulting in increased warranty costs and decreased customer satisfaction.

In this thesis we look at diagnosis in the context of integrated architectures, since integrated architectures, as already partly deployed in avionics, promise massive cost savings due to the multiplexing of hardware resources among different application subsystems. Furthermore, the resulting reduction of wiring and connectors results in dependability improvements. For this reason, integrated architectures are gaining more and more momentum also in the automotive domain in order to resolve the pending "one function - one control unit" problem.

In order to cope with industry demands on diagnosis, as part of this thesis a framework is introduced for error detection and subsequent identification of the field replaceable units causing system malfunction. Such a framework provides generic diagnostic services to be parameterized according to the developer's needs and thus avoids costly redesign of individual diagnostic solutions at application level.

Based on industrial requirements, a maintenance-oriented fault model is presented. This fault model takes the component-based nature of today's distributed embedded systems into account. According to this model each experienced failure is classified with respect to the field replaceable units of the system and stops the recursion of the fault-error-failure chain at a level suitable for maintenance. The pivotal strategy of the diagnostic architecture is the establishment of a holistic view on the system by operating on the distributed state established via the underlying core services. To capture the characteristics of the fault-induced distributed state changes in the value, space, and time domain, we introduce the concept of out-of-norm assertions in order to discriminate between different types of faults that are affecting the operation of the distributed system. In the introduced framework, for the specification of the diagnostic mechanisms timed automata are used.

For the transport of diagnostic information a dedicated virtual network is established. The key advantage of such an encapsulated network dedicated to diagnosis is the fact that real-time traffic is not compromised in any way and no probe effects can be introduced. Based on the maintenance-oriented fault model this diagnostic information is then processed to determine the health status of the field replaceable units of the system. The result supports the service technician in the decision process whether a field replaceable unit remains in the system or will be replaced.

The proposed methods are implemented and evaluated in a prototype setup of the integrated time-triggered architecture. In this prototype setup the feasibility of the proposed architecture is investigated. Furthermore, fault injection campaigns have been carried out, to analyze the effects of faults on the distributed state of state time-triggered core network and to validate claims of the diagnostic architecture.

Diagnose und Wartung in einer integrierten zeitgesteuerten Architektur

Der Einsatz von Elektronik in der Automobil- und Flugzeugindustrie hat zu zahlreichen Innovationen im Bereich der Sicherheit, der Antriebstechnik, des Passagierkomforts sowie zu einer Reduktion der Systemkosten geführt. In den letzten Jahren hat allerdings die Entwicklung effizienter on-board Diagnosesysteme dem technischen Fortschritt auf anderen Gebieten nicht Stand gehalten. Mit dem Anstieg von transienten Fehlerraten von IC's aufgrund der höheren Integration hat diese Entwicklung zu einer signifikanten Erhöhung von nicht reproduzierbaren Fehlern mit weit reichenden wirtschaftlichen Konsequenzen geführt.

In Rahmen dieser Arbeit wird Diagnose im Kontext von integrierten Architekturen untersucht, da integrierte Architekturen, wie bereits teilweise in der Flugzeugindustrie eingesetzt, massive Kosteneinsparungen durch das Multiplexen von Hardware Ressourcen zwischen den verschiedenen Applikationssubsystemen erlauben. Weiters ermöglicht die Reduktion von Steckern und Kabelverbindungen eine erhebliche Verbesserung in Hinblick auf die Zuverlässigkeit des Systems. Aus diesem Grund gewinnen integrierte Systemarchitekturen auch in der Automobilindustrie immer mehr an Bedeutung, insbesondere um das vorherrschende "eine Funktion - ein Knoten" Problem zu lösen.

Um den industriellen Diagnoseanforderungen zu genügen, wird als Teil dieser Arbeit eine Rahmenumgebung vorgestellt, welche die online Fehlererkennung und Identifikation der im Feld austauschbaren Systemkomponenten unterstützt. Es werden generische Diagnoseservices zur Verfügung gestellt, die nach den Anforderungen der Entwickler parametrisiert werden können. So wird ein aufwendiges und kostenintensives Re-design von individuellen Diagnoselösungen auf Applikationsebene vermieden.

Ein zentrales Element der Arbeit ist die Definition eines wartungsorientierten Fehlermodells. Dieses Fehlermodell berücksichtigt die typische Komponentenstruktur heutiger verteilter eingebetteter Systeme. Nach diesem Modell wird jeder aufgetretene Fehler nach den austauschbaren Einheiten des Systems klassifiziert. Damit wird die Rekursion der "Fault-Error-Failure" Kette auf einer für die Wartung passenden Ebene gestoppt. Die Schlüsselstrategie der vorgestellten Diagnosearchitektur ist die Operation auf dem verteilten Zustand des Systems. Um die Charakteristik von fehlerinduzierten Zustandsänderungen in Wert, Zeit, und Raum einzufangen, wird das Konzept der Out-of-Norm Assertions eingeführt, um zwischen den verschiedenen Fehlerklassen zu unterscheiden. Für die Spezifikation der Diagnosemechanismen werden Zeit-Automaten benutzt.

Zum Transport der Diagnosenachrichten wird ein der Diagnose a priori zugeordnetes virtuelles Netzwerk verwendet. Der Hauptvorteil eines solchen gekapselten Netzwerkes besteht darin, dass andere Echtzeitnachrichten nicht beeinflusst werden und somit kein "Probe Effekt" entstehen kann. Die Diagnoseinformationen werden anschließend nach dem wartungsorientierten Fehlermodell verarbeitet, um den Zustand der austauschbaren Systemkomponenten festzustellen. Das Resultat des Analyseprozesses dient dem Wartungstechniker im Entscheidungsprozess, ob eine Systemkomponente getauscht werden muss.

Die entwickelten Methoden wurden im Rahmen eines Prototyps der integrierten zeitgesteuerten Architektur implementiert und evaluiert. In diesem Prototypenaufbau wurde die Durchführbarkeit der vorgeschlagenen Architektur untersucht. Weiters wurden Fehlereinstreuungsexperimente ausgeführt, um die Effekte von Fehlern auf den verteilten Zustand des unterliegenden zeitgesteuerten Systems zu analysieren und Behauptungen der Diagnosearchitektur zu validieren.

Contents

List of Figures

Chapter 1

Introduction

There is a significant trend in the automotive and avionics industry to increase the number of electronic devices in order to provide a functionality that goes beyond common mechanic/hydraulic systems. The reduction of cost, increased safety and reliability, reduced complexity, and enhanced quality of control are among the primary objectives of replacing conventional subsystems with electronic ones. Fly-by-wire airplanes like the Boeing 777/787 or the Airbus A320/330/340/380 series aircrafts show impressively the benefits and capabilities of state-of-the-art electronic devices and architectures [MS03]. Also in the automotive domain without the use of electronics in modern cars many features like anti-lock braking, stability programs, airbags, cruise control, intelligent motor management and many more would be impossible. With decreasing hardware costs and increased reliability of electronic devices this trend will continue even more in the future. For example, in the automotive domain it is estimated that more than 80% of all innovations now stem from electronics [LH02]. According to [Ber02] in the 2001 model year, electronics accounted for 19 percent of a mid-sized vehicle's cost, while the cost of the electronics in luxury vehicles can amount to more than 23 percent of the total manufacturing costs [LHD99, LH02].

However, despite all the benefits it is important to state that with the increasing use of electronic devices in transportation systems the likelihood of malfunctions and thus the numbers of defective electronic components will also increase.

Originally developed to provide simple open/short circuit and abnormal voltage level detection mechanisms, electronic diagnosis evolved into an integral part of every automobile. All modern cars are equipped with On-Board Diagnosis (OBD) systems (OBD-II in USA or EOBD in Europe). OBD, originally developed to continuously monitor the emissions of a car, provides now almost complete engine diagnosis and also monitors parts of the chassis, body electronics, and the control network of the vehicle [OM02].

However, the development of effective diagnostic systems has stayed behind the recent increase of electronic systems in modern cars. One reason for the diagnostic deficiencies of modern OBD systems is the fact that diagnosis is often treated

as add-on to communication systems rather than an integral part of the architecture [SSW00]. Consequently, the problem of the identification of faulty Electronic Control Units (ECUs) is one of the predominant challenges that needs to be solved.

Though the breakdown logs of the ECUs inform the service technician about detected errors within the system, they do not assist the technician adequately in the identification process [Bar01]. Thus, fully functional units are replaced, or even worse, faulty ECUs remain unchanged in the system. These diagnostic deficiencies will become more and more obvious when X-by-wire solutions will be subject to mass production [Bre01]. Since a mechanic at a service station is no specialist in automobile electronics, the diagnostic system of the car must provide all necessary information that allows maintenance of faulty components. For this reason it must be possible in modern automotive electronic architectures to trace an entry in a breakdown log back to its source. If this is not possible, as a consequence, fully operational units will be replaced by mistake.

Many deployed OBD systems analyze the internal state of a component (e.g., plausibility checks) by applying embedded assertions in the application software in order to identify component errors. Assertions are a powerful and accepted mechanism in helping in the detection of application errors. However, such assertions operate in general only on the internal state of components. The inability to trace correlated failures of the nodes of a distributed system makes diagnosis prone to misjudgement about transient faults affecting the system. These so-called *cannot duplicate failures* frequently result in the replacement of operational Field Replaceable Units (FRUs) [TAP02, MRS+02, Scu98]. As a consequence, these spurious failures have a lasting effect on the customer's trust in the product and the reputation of the manufacturer.

In contrast to permanent component failures that have been massively reduced by improvements of the manufacturing process [PMH98], transient failures in electronic systems impose a serious problem to electronic system designers and manufacturers. The primary cause for the significant increase of soft error rates are shrinking geometries, lower power voltages and higher frequencies [Con02]. Furthermore, the likelihood of transient failures is also growing due to semiconductor process variations and manufacturing residuals. Consequently, what is required in distributed embedded systems, is a diagnostic subsystem with detection mechanisms focussing on transient failures. In particular, from a maintenance perspective, the most important diagnostic objective is the discrimination between transient failures induced from external and internal faults to put an end to unnecessary component replacements.

Today, in the automotive domain we find electronic system architectures that cannot be classified as either completely federated nor integrated. Typically, electronic systems in the car do not provide mechanisms to share an ECU among multiple applications and thus adhere to the federated systems design principle. However, for economic reasons, distributed application subsystems share typically the same communication infrastructure (e.g., a CAN network for the interconnection of the comfort

electronics in a car). However, this system design has the significant drawback of increased complexity. Since reasoning of the behavior of a particular application requires the understanding of all other applications participating in the communication, this design makes it hard to comprehend the emerging interdependencies between applications. Furthermore, in case of failure, missing error containment capabilities make system integration and diagnosis a challenging and cost intensive task. Since integration responsibilities cannot easily be assigned, even the launch of new car models can be delayed. The "1 Function - 1 ECU" design philosophy that is characteristic for this system design approach leads to a dramatic increase of the numbers of ECUs due to the increased functionality of today's cars required by the expectations of the customers. For example, luxury cars can have more than 70 ECUs [Dei02]. Since such a high number imposes problems with respect to architecture complexity, wiring, mounting, resource duplication and many others, a reduction of the number of ECUs is of great interest.

Integrated architectures as already partly deployed in avionics [Aer91], promise also massive cost savings by addressing problems the automotive industry is currently facing. Integrated architectures are characterized by the possibility to share components among multiple applications. In addition, integrated systems permit an optimal interplay of application subsystems, reliability improvements with respect to wiring and connectors, and overcome limitations for spare components and redundancy management. An ideal future system architecture would thus *combine the complexity management advantages of the federated approach, but would also realize the functional integration and hardware benefits of an integrated system* [Ham03, p. 32]. The challenge is to devise an integrated architecture that provides a framework with generic architectural services for integrating multiple application subsystems within a single, distributed computer system, while retaining the error containment and complexity management benefits of federated systems.

1.1 Objectives and Contribution of the Thesis

The main objective of this thesis is to devise generic diagnostic architectural services as part of a certifiable integrated architecture suitable for both ultra-dependable applications and those having less stringent dependability requirements in favor of increased flexibility. In the following, we will identify the primary objectives and contributions of the thesis.

Integrated Solution

The DECOS architecture, as being developed in the course of the Sixth European Framework Programme, is designed to combine the benefits of the federated and integrated system design. The DECOS architecture [KOPS04] offers a framework for the development of distributed embedded real-time systems integrating multiple applications with different levels of criticality and different requirements concerning the

underlying platform. Any time-triggered architecture that provides four basic services, i.e. predictable message transport, fault-tolerant clock synchronization, strong fault isolation, and consistent diagnosis of failing nodes, can be used as the core architecture for a DECOS integrated system. Based on these core services, high-level services are realized that are specific to an encapsulated application subsystem (e.g., brake-by-wire subsystem) and facilitate the development of applications by providing a stable interface to commonly used functionality thus reducing the complexity of application development. The encapsulation of application subsystems also facilitates independent development and system integration.

As part of the design of this integrated system architecture, a diagnostic infrastructure is devised that in contrast to many addendum solutions deployed today allows evolving beyond "best guess maintenance". Our solution provides a framework for both systemic and application-specific diagnosis. While systemic diagnosis focusses on the assessment of the health status of the underlying platform (e.g., physical components, connectors), application-specific diagnosis aims at revealing application inherent faults such as software and actuator/sensor faults. Since the DECOS architecture sharply separates application from architecture level, the systemic checks can be deployed independently from a particular application. By precisely defining the interface state of the applications, applications diagnosis can be handled in a generic manner outside the application functionality, as required by today's industrial demand for intellectual property protection (i.e. no modification of the application source code). This way only the application inherent complexity must be dealt with during application development but no additional complexity is introduced by the diagnostic subsystem.

A dedicated virtual network, i.e. an overlay network on top of the time-triggered physical core network, is used for the transport of diagnostic information. This way no interference with the real-time services or a probe effect due to hardware failures (e.g., loose contacts) can be introduced that might mislead the analysis process. Furthermore, such a pure virtual solution has also the benefit of keeping the associated costs to a minimum.

Another important fact is, that the proposed diagnostic solution does not restrict the choices of implementation of the analysis subsystem. Either a central or a distributed solution can be realized.

A Fault Model for Maintenance

In order to tackle prevalent maintenance problems a maintenance-oriented fault model needs to be devised that takes the component-based nature of distributed systems into account. According to this model each experienced failure is classified according to the FRUs of the system. In integrated architectures, a one to one mapping between applications and physical components is no longer feasible. This has to be taken into account in a maintenance-oriented fault model that establishes a basis for a better understanding of the diagnostic problems of modern distributed systems

and introduces a maintenance-specific fault classification. Consequently, the model stops the recursion of the fault-error-failure chain at a level suitable for maintenance. According to this model, the diagnostic analysis algorithms of the integrated architecture can assess the health state of each FRU and determine whether a change of a particular FRU can eliminate an experienced problem. The maintenance-oriented fault model is based on related literature, field data, and discussions with industrial partners.

Furthermore, this model forms also the conceptual foundation for the validation of the diagnostic mechanisms, e.g., by means of fault injection experiments. By injecting representative faults for every devised fault class (e.g., Electromagnetic Interference (EMI) for external transient faults) the accuracy of the detection and analysis mechanisms can be evaluated.

Improved Accuracy of Diagnosis

Currently, industry has significant problems detecting and identifying electronic devices that cause system failures in electronic systems. This so called *Trouble Not Identified (TNI) phenomenon* is an increasing problem in automotive and avionic electronics with major economic implications [TAP02]. The lack of information provided by currently deployed OBD systems often results in unnecessary replacements of working components [MRS$^+$02, TE01].

By operating on the distributed state of the system instead of the component local state correlation of detected failures is possible and thus an improved accuracy of the assessment process can be achieved. The exploitation of the knowledge of the physical and functional structure of the integrated system allows a finer granularity of the analysis that would not be possible in a federated computer system. By overcoming the "1 Function - 1 ECU" design, a discrimination between hardware and software is feasible. Furthermore, the strong fault isolation capabilities of the time-triggered core architecture simplify to answer the question which part of the system is to blame for the experienced malfunction. In combination with the availability of a global sparse time base, a correlation of failures based on their timestamps can be detected. The inclusion of the temporal, spatial and value domain ultimately helps to answer the question of the mechanics at the service station whether to remove a FRU or to leave it in the system.

Advanced Maintenance Strategies

In both avionics and in the automotive domain manufacturers envisage a shift from traditional corrective to preventive maintenance strategies to reduce costs and to provide optimal availability of the systems. As a prerequisite for the realization of preventive maintenance strategies, diagnostic mechanism are needed to judge about the health status of each replaceable part of the system. If advanced maintenance techniques like Condition-Based Maintenance (CBM) are envisaged, then new assessment

techniques for the condition of electronic devices need to be identified [PP01, TS01]. In order to adopt CBM for electronic systems suitable indicators for degradation or wearout must be identified and analyzed to detect deviations from sound operation. Since the increase of the transient failure rate is an acknowledged indicator for premature electronic component wearout [Con02], the integrated diagnostic architecture must be designed to continuously monitor the distributed state of the system, and thus detect and record any transient anomaly. This is especially important in case of fault-tolerant systems, where a level of redundancy must be maintained to ensure safe operation even in the case of failure (according to the fault hypothesis). Consequently, monitoring of the deployed fault-tolerance mechanisms is mandatory and a replacement of defective parts by the service technician must be enabled without making the owner of the system insecure, i.e. keep the user's trust in the product.

1.2 Structure of the Thesis

The thesis is structured as follows. In Chapter 2 an elaboration on the concepts of time, state, component, interface and dependability is presented. Related work in the field of diagnosis and maintenance is presented in Chapter 3. Both avionics and automotive diagnosis and maintenance techniques are discussed, in particular the OSEK and ARINC strategies. Furthermore, the increase of the transient failure rate of electronic devices and its consequences are subject to investigation. In addition, the related work section covers an overview on prevalent analysis techniques. The DECOS integrated architecture is presented in Chapter 4. We first discuss the structure of the architecture, core and high-level services, and detailed component structure. In addition, we present the design flow and the underlying dependability model including the fault hypothesis for the integrated DECOS architecture. The integrated diagnostic architecture is introduced in Chapter 5. We elaborate on the maintenance-oriented fault model, introduce out-of-norm assertions as the primary diagnostic mechanism operating on the distributed state, and discuss the virtual diagnostic network and analysis subsystem. In Chapter 6 a prototype implementation of the DECOS architecture is presented. We describe the prototype platform and the implementation choices. For the implementation the time-triggered protocol (TTP) is deployed as the time-triggered core architecture. Furthermore, we show how the different detection and analysis techniques are realized and discuss the performance of some of the implemented analysis algorithms. In Chapter 7 we present selected results from fault injection campaigns. Finally, the thesis is concluded in Chapter 8.

Chapter 2

Basic Concepts

The concepts of time, state, component, interface and dependability are fundamental in the design of dependable embedded real-time systems. This chapter elaborates on the close relationship between these concepts that constitute the conceptual foundation of this thesis.

First, we discuss the concept of time, since time must be treated as first-order quantity in the design of a real-time system. The following section provides an overview of the meaning of state and the close relationship between time and the concept of state. Furthermore, we elaborate on the difference between system components and software components as the building blocks of dependable distributed systems. Since both components interact with the environment via interfaces the subsequent part discusses software and messages interfaces. This chapter is concluded by presenting the concepts of dependability.

2.1 Time

Whitrow [Whi90] notes that our actual experience of time can be analyzed in terms of two fundamental relations: simultaneity and temporal order (or precedence). In respect of these, any event is judged to be either simultaneous with or else either earlier than or later than any other event [Whi90, p. 207]. Russell defined (in 1914) an instant as a set of events, any two of which are simultaneous and there is no other event which is simultaneous with them all. The existence of so defined instants was assumed. An event is said to be *at* a particular instant when it is a member of the set defining that instant. Temporal order of instants is then defined by stipulating that one is earlier than another if there is some event at the former that is earlier than some event at the latter. If neither instant is earlier than the other, then they are simultaneous (identical). The continuum of real time can be modeled by a directed timeline [Wie14, Rus36] consisting of an infinite set T of instants with the following properties [Whi90]:

1. T is a simply ordered set, that is, if p and q are any two instants, than either p is simultaneous with q, or p precedes q, or q precedes p, and these relations are mutually exclusive. Furthermore, if p precedes q and q precedes another instant r, then p precedes r, and q is said to be between p and r.

2. T is a dense set. This means, that, if p precedes r, there is at least one q which is between p and r.

3. T satisfies Dedekind's postulate, namely if T_1 and T_2 are any two non-empty parts of T such, that every instant of T belongs either to T_1 or T_2 and every instant of T_1 precedes every instant of T_2, then there is at least one instant t such that any instant earlier than t belongs to T_1 and any instant later that t belongs to T_2.

This prevalent mathematical picture of "standard time", namely that of a set of *instants* with a temporal precedence order $<$ satisfying certain obvious conditions, can also be found in [vB83]. These conditions are:

1. Transitivity

2. Irreflexivity

3. Linearity

4. Eternity $(\forall x \exists y : y < x, \forall x \exists y : x < y)$

5. Density $(\forall x, y : x < y \rightarrow \exists z : x < z < y)$

The most obvious models that satisfy these axioms are the rational numbers $\mathbb{Q}$ and the real numbers $\mathbb{R}$. These sets are dense, i.e. for any two numbers there is another number that lies between them. When replacing density by *discreteness* the integers $\mathbb{Z}$ are the standard model of time [Mat88].

The abstract mathematical idea of time as a geometrical locus – the so-called spatialization of time – is one of the most fundamental concepts of modern science. Its psychological origin lies in our intuitive conception of time as one-dimensional. Our idea of time is thus directly linked with the fact that the process of thinking has the form of a linear sequence [Whi90]. This linear sequence consists of discrete acts of attention. Consequently, in the first instance, time is more naturally associated with counting, and hence with numbers, than with the linear continuum of geometry. The peculiarly close relation between time and counting has been emphasized both by philosophers of time and by philosophers of mathematics [AriCE].

An essential difference between concurrent models of computation is their modeling of time [Lee01]. Some models are very explicit by taking time to be a real number that advances uniformly, and placing events on a time line or evolving continuous signals along the time line. In other models time is taken to be discrete, while other

models, even more abstract, take time to be merely a constraint imposed by causality [Lee01]. In the context of real-time systems a time model based on Newtonian time seems to be the best choice [Kop97].

Since a behavior, i.e. *the temporal sequence of send operations of a system in relation to its previous receive operations, and any internal state that it retains* [GIJ+02, p. 83] can only be associated with a system if some notion of time is taken into account [Kop97] and time is also important for the introduction of the concept of a failure, organizing further definitions around a directed timeline that extends from the past into the future is justified [Kop97]: A cut of the timeline is an instant. Any occurrence that happens at an instant is called an *event*. Information that describes an event is called event information. Event information is non-idempotent and requires exactly-once semantics when transmitted to a consumer. The present instant, *now*, is a very special instant that separates the past from the future An *interval* on the timeline is defined by two instants, the start event and the terminating event of the interval. Any property of an object that remains valid during a finite duration is called a state attribute and the corresponding information state information. State information is idempotent and requires at-least-once semantics when transmitted to a consumer. A change of state is an event. An observation is an event that records the state of an object at a particular instant, the point of observation. An event observation can be expressed by the atomic triple $\langle \text{name}, \text{value}, \text{time} \rangle$.

Sparse Time Base

If the time-base of a distributed system is dense (the events are allowed to occur at any instant of the timeline), then it is in general not possible to generate a consistent temporal order on the basis of the time-stamps [Kop97]. Due to the impossibility of synchronizing clocks perfectly and the denseness property of real time, there is always the possibility that a single event is time-stamped by two clocks with a difference of one tick. By introducing the concept of a *sparse time base* this problem can be solved [Kop92]. In the sparse time model the continuum of time is partitioned into an

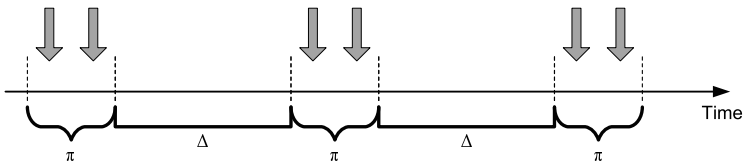

Figure 2.1: Sparse Time Base

infinite sequence of alternating durations of activity (π) and silence (Δ) as shown in Figure 2.1. Thereby, the occurrence of significant events is restricted to the activity intervals of a globally synchronized action lattice. In this time model, the costly execution of agreement protocols can be avoided, since every action is delayed until the next lattice point of the action lattice [Kop92].

2.2 State

The notion of state is widely used in computer science, but when it comes to the exact definition of state, difficulties arise. Depending on the context of the scientist the definition varies. For example software engineers tend to describe the state of a system using the variables of the objects; system theoreticians try to explain the state of a system independently of its context.

2.2.1 The Concept of State

In system theory, the notion of state is fundamental for the investigation of complex systems. In abstract system theory, the notion of state is introduced in order to separate the past from the future (decoupling). *The idea is that if one knows what state the system is in, he could with assurance ascertain what the output will be* [MT89, p. 45]. Hence, *the state embodies all past history of the given system* [MT89, p. 45]. Apparently, this definition of state by Mesarovic and Takahara is only meaningful, if the notion of past and future (time) is relevant for the considered system.

Other system theoreticians are in agreement with the above presented definition expressed in terms of time. For instance, Zadeh states informally that the *notion of state of a system at any given time is the information needed to determine the behavior of the system from that time on* [Zad69, p. 3].

By contrast, in the field of software engineering and object oriented programming languages the notion of state is closely related to the attributes of an object. In the Java programming language tutorial the state of an object is described as *everything that the software object knows* [CWH00]. This is consistent with the view of Szyperski, who explains in the context of Java that *all state resides in the attributes of classes* [Szy98, p. 219]. He further argues that a software component (e.g., a library) can make part of its state observable by exporting variables or inspection functions.

In the context of theoretical computer science Clarke defines the state as a *snapshot or instantaneous description of the system that captures the values of the variables at a particular instant of time* [CGP99, p. 13].

Another definition of the state of a system can be found in the DSoS (Dependable Systems of Systems) conceptual model. The model uses the concept of an internal data structure that *synthesizes all cumulative effects of all receive operations at all input interfaces between the startup of the system and this given instant* [GIJ+02, p. 85].

The *declared state* is the state of a subsystem, which is considered as relevant by the system designer for the future behavior of the subsystem (forward view). The *interface state* contains the history of the component that is relevant for the future behavior of the component as seen from this interface. Interface state is defined between the intervals of activity on the sparse time base [Kop92]. Interface state is a subset of the state of the component and should be accessible from the interface.

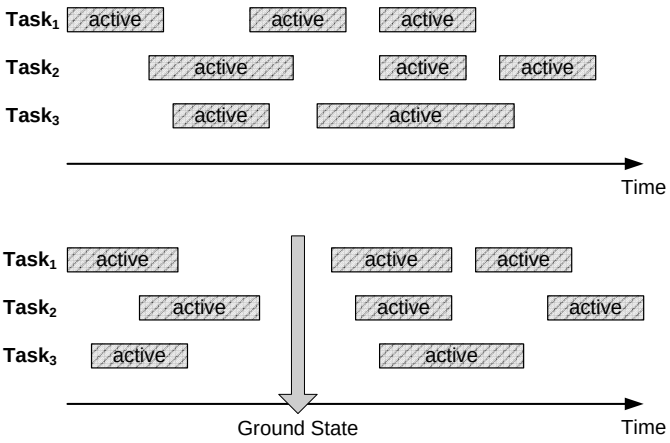

Figure 2.2: The Concept of Ground State

2.2.2 Ground State

The ground state of a node in a distributed system at a given level of abstraction is defined as a state where no task is active and where all communication channels are flushed, i.e. *there are no messages in transit* [AKC90, p. 1]. Consider a node that contains a number of concurrently executing tasks that exchange messages with each other and with the environment of the node. If a level of abstraction that considers the execution of a task as an atomic action is chosen, and the executions of the tasks are asynchronous, then the situation depicted in the upper section of Figure 2.2 can arise; at every point in real time, there is at least one active task, thus, implying that there is no point in real time when the ground state can be defined. In the lower part of Figure 2.2, there is an instant where no task is active and all the communication channels are empty, i.e. where the system is in the ground state. If a system is in the ground state, then the internal state is contained in its data structures and the program counter. The reintegration after a failure is simplified if a system periodically visits a ground state that can be used as a reintegration point [Kop97].

2.3 Component

A standard problem solving technique is the division of complex systems into nearly-independent subsystems [Sim96]. This intuitive approach to manage complexity can be found in many engineering disciplines, where large systems are assembled from prefabricated components with known and validated properties. A component is regarded as a self-contained subsystem that can be used as a building block in the design of a larger system. An example of such components is the engine, gearbox, or wheel suspension in an automobile. The component can have a complex internal structure that is neither visible, nor of concern, to the user of the component.

An ideal component should maintain its encapsulation when used in a larger context [KS03a].

One of the most apparent problems in the domain of computer systems relates to constructive design of large systems out of independently developed pre-validated components. In the following we investigate two different types of components and their definitions, namely *system components* and *software components*. At first various definitions of software components are presented. Most of these definitions are stated in the field of Component-Based Software Engineering (CBSE), which is concerned with the rapid assembly of software systems from pre-built components of independent vendors. The second part gives insight into the concepts of system components. A system component is a self-contained composite HW/SW subsystem that can be used as a building block in the design of a larger system. In contrast to software components, system components are time-aware and thus allow a meaningful definition of state.

2.3.1 Software Component

In Component-Based Software Engineering (CBSE), which is concerned with the rapid assembly of systems from components [BW98], the opaque implementation of functionality is a fundamental idea that allows third-party composition. A component implements one or more interfaces that are imposed upon it, which reflects that the component satisfies certain obligations, so called contracts [HHG90, Hol92]. These contractual obligations ensure that independently developed components obey certain rules so that components interact (or cannot interact) in predictable ways, and can be deployed into standard build-time and run-time environments. A component-based system is based upon a small number of distinct component types, each of which plays a specialized role in a system and is described by an interface [BBBCD00]. CBSE usually sees a component as a replacement (reusable) unit like a commercial off-the-shelf commodity. The components with certified properties provide the basis for predicting the properties of systems built from components.

Various definitions of component in the context of CBSE can be found [KJH00]. Microsoft, for instance, defines a component as a piece of software, which is offering a service [WK94]. As defined by the OMG, a component is the minimal piece of functionality in a system or subsystem that can be removed without affecting the integrity of the system or subsystem [Was95]. Cuipke and Schmidt [CS96] concentrate on the context-independence of a component and D'Souza [DW99] defines components as reusable parts of software that can be adapted but not modified. Herzum and Sims [HS00] emphasize the autonomous deployment and collaboration of components. Wijnstra describes components as *units containing reusable functionality with explicit interfaces* [Wij01, p. 27].

Szyperski [Szy98] defines software components as *binary units of independent production, acquisition, and deployment that interact to form a functioning system* [Szy98, preface]. Insisting on independence and binary form is essential to allow

for multiple independent vendors and robust integration. Furthermore, Szyperski identifies three major forces in the field of CBSE: the OMG (corporate enterprise perspective) with CORBA-based standards, Microsoft (desktop perspective) with COM-based standards and Sun (Internet perspective) with Java-based standards. From an economic point of view Szyperski remarks that, compared with specific solutions to specific problems, components need to be carefully generalized to enable reuse in a variety of contexts. Solving a general problem rather than a specific one takes more work. Components are thus viable only if the investment in their creation is returned as a result of their deployment. In traditional software engineering processes, module development and testing is followed by systems integration and testing. This approach leaves room for errors that are merely a result of composition and which are not apparent at the level of individual components, but which should be detected during systems integration and testing. With third-party integration, the situation becomes more difficult, as integration testing of modules from different sources needs to be addressed. The resulting problem in an open market of independent component developers is the fact that the set of possible combinations is not even known to any one of the involved parties. Szyperski also addresses component safety as an important issue, namely that a component must not violate system-wide rules.

Kruchten's definition, though stated in the context of software component design, is a more general one. He defines a component as a *non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces* [Kru98, p. 1]. In a closer examination the properties of the definition are explained. The author states that a component is substitutable for any other component that realizes the same interfaces. Logical and physical cohesiveness of a component denotes a meaningful structural and/or behavioral part of a larger system. Furthermore a component represents a fundamental building block out of which systems can be designed and composed. Conformity of a component to a given interface means that it satisfies the contract specified by that interface and may be substituted in any context wherein that interface applies.

2.3.2 System Component

In contrast to the view of software engineering Kopetz defines a system component as a *self-contained composite hardware/softwares subsystem that can be used as a building block in the design of a larger system* [KS03a, p. 3]. In the context of embedded real-time systems a complete node seems to be the best choice for a component [Kop98], since the component-behavior can then be specified in the domains of value and time. Thus, a component is considered to be a self-contained computational element with its own hardware (processor, memory, communication interface, and interface to the controlled object) and software (application programs, operating system), which interacts with its environment by exchanging messages across Linking Interfaces (LIFs).

In order to bring the advantages of component-based software engineering to embedded systems, the special domain characteristics of embedded systems have to be taken into account [MSZ01]. An ideal component should be an autonomous unit that maintains its encapsulation. Kopetz proposes various properties of such an ideal component, namely [Kop00]:

A unit of service provision: A component must be a unit of service provision by offering services to the component's environment across a real-time service interface. In a distributed real-time system, the service consists of the timely processing and provision of the requested information.

A unit for validation: The validation of the proper operation of a component in the value domain and in the temporal domain must be possible in isolation. The preconditions for the correct operation of the component, both in the domains of time and value, must be precisely specified in the interface specifications.

A unit of error containment: All errors that occur inside a component must be detected before the consequences of these errors propagate across a component's interface. Otherwise, a defective component can falsify the operation of other components by the provision of corrupted output data across the component interface.

A unit for reuse: A component should be a unit for reuse. This requires that the component has standardized interfaces to support the integration of the component in various system contexts.

A unit of design and maintenance: A component should be a unit of design and maintenance. It is well known that system structures evolve along organization structures. If the work output of an organizational group is a nearly autonomous subsystem with well-specified interfaces, then the management of this group is simplified. The error containment boundaries around a component reduce the possibility of unforeseen consequences of software maintenance actions.

According to Kopetz an architecture is said to be *composable* with respect to a specified property if *the system integration will not invalidate this property once the property has been established at the subsystem level* [Kop97, p. 34]. Examples of such properties in the context of distributed real-time systems are timeliness and testability. The components are characterized by their physical parameters and the services they provide across well-specified interfaces. In a composable architecture, this integration should proceed without unintended side effects. From the point of view of the analysis of a composable architecture, it is reasonable to distinguish between the two service classes of an integrated distributed control system [Kop01, p. 227]: *prior services* and *emerging services*. A prior service of a component is the specified service that has been developed independently from the system it will be deployed. Prior services can be validated at component level and thus be available

prior to the integration of the component into an integrated control system. In contrast to emerging services that result from the integration of components into a system. Such an integration generates new services that are more than the sum of the prior services.

In a distributed system, a given set of components compositionally realize emergent services by exchanging messages across LIFs. The emergent services refer to those properties of the system of components that are not explicitly properties of the components themselves, but come into existence by the interactions among the components across LIFs. Therefore, the communication system plays a central role in determining the composability of distributed computer architecture. For an architecture to be composable, it must adhere to the principles with respect to the real-time service interface as introduced in [KO02].

2.4 Interface

Usually an interface is referred to as a common boundary between two subsystems. Since *architecture design is primarily interface design, the most important phase in the design of large system architecture is the layout and placement of the interfaces* [Kop97, p. 77]. A correctly designed interface provides understandable abstractions, which capture the essential properties of the interfacing subsystems and hide irrelevant details (control, temporal, functional, and data properties).

2.4.1 Software Interfaces

In software engineering, interface abstraction provides a mechanism to control the dependencies that arise between modules in a program or system [BBBCD00]. The theory is that information hiding makes modules substitutable (for example, with new versions of a component), and hence makes systems easier to change, at least insofar as module substitution is concerned. Bachmann et al. [BBBCD00] emphasize the importance to distinguish between abstract interfaces (those that are described independently of any implementation) and bound interfaces (those that are associated with an implementation) for certification, composition, and system analysis.

The idea of an *interface contract* has become prominent in CBSE research literature. A contract, as introduced in [HHG90] and extended in [Hol92], is a descriptive formal (or semiformal) language by which clauses in a programming context may be written to explicitly specify interactions among groups of objects (participants). These interface contracts are between two or more parties, which often negotiate the details of a contract before becoming signatories. Contracts prescribe normative and measurable behaviors on all signatories and can not be changed unless the changes are agreed to by all signatories. A component contract specifies a pattern of interaction rooted on that component. The contract specifies the services provided by a component and the obligations of clients and the environment needed by a component to provide these services. Contracts shift the focus from specification of

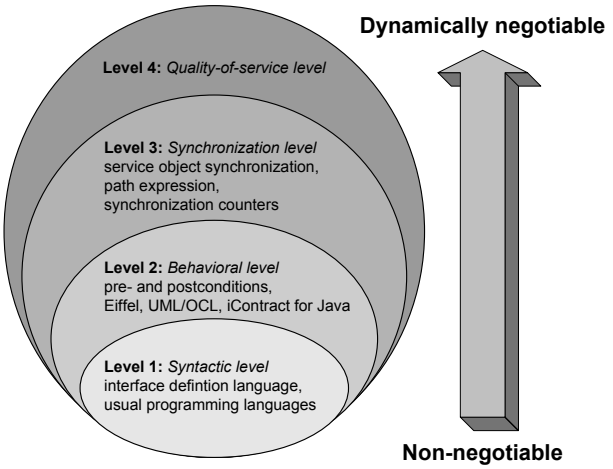

Figure 2.3: The Four Contract Levels

components to specification of patterns of interactions, and the mutual obligations of participants in these interactions.

Beugnard et al. [BJPW99] identifies four levels of contracts in the software component world (functional versus extra-functional properties). The first level, syntactic contracts, is required simply to make the system work (interface definition language, usual programming languages). The second level, behavioral contracts, improves the level of confidence in a sequential context (pre- and postconditions). The third level, synchronization contracts, improves confidence in distributed or concurrency contexts. The fourth level, quality-of-service contracts, quantifies quality of service and is usually negotiable. This model is depicted in Figure 2.3.

This is in agreement with the view of Brown and Wallnau. They state that many researches in CBSE suggest that components should implement two interface types: *A functional one that reflects the component's role in the system, and an extrafunctional one that reflects the component model imposed by some underlying component framework* [BW98, p. 40]. Nevertheless the majority of commercial off the shelf software components are specified only with functional attributes in their interfaces [BRO+02]. Brahnmath et al. address the issue of interface classification by proposing a quality of service catalog for heterogeneous software components. By introducing a QoS metric that addresses functional and non-functional requirements, system developers are able to validate and verify the claims of the component developer.

Szyperski describes interfaces as the *means by which components connect* [Szy98, p. 40]. One can *consider the interface of a component to define the component's access points* [Szy98, p. 34]. The required interfaces of a component will specify what deployment environment (context dependencies) will be needed such that the component will function according to its specification. He further argues that a useful way to view interface specifications is as contracts between a client of an interface and a provider of an implementation of the interface. *The contract states what the*

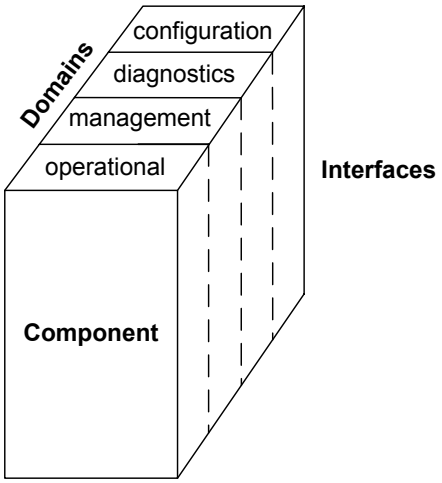

Figure 2.4: Structuring of Interfaces

client needs to do to use the interface [Szy98, p. 43]. It also states what the provider has to implement to meet the services promised by the interface, and which non-functional requirements (e.g., time, resources) must not be neglected. However it is important that a contract does not overspecify a situation.

Kruchten defines an interface as a *collection of operations that are used to specify a service of a component* [Kru98, p. 1]. An interface focuses upon the behavior, not the structure, of a given service while offering no implementation for any of its operations. In the scope of software engineering an interface defines a service that is implemented by a class or a component. As such, an interface spans the logical and physical boundaries of a system. One or more classes (which are likely a part of some component subsystem) may provide a logical implementation of a given interface; one or more components may provide a physical packaging that conforms to that same interface.

Wijnstra identifies several important interface characteristics. Interfaces *provide access points to clearly defined functionality for use by other components* [Wij01, p. 27]. The components must implement interfaces in their entirety (no optional methods), the interfaces should be stable but the implementation can be flexible. Also the same interface may be implemented by multiple components. Wijnstra emphasizes the close relation to components by indicating two types of interfaces, namely:

Provided Interface: the component guarantees that it will implement the functionality associated with the interface

Required Interface: the component accesses functionality through this interface and relies on the functionality to be implemented outside the component.

The basic idea in the approach presented by Ran and Xu *is the belief that just as components are structured as compositions of lower level components, interfaces*

must be structured as composition of lower level interfaces [RX97, p. 31]. Their approach is justified by the observation that interface elements are not structured or abstracted but are simply propagated as such through component aggregation hierarchies in contrast to the common practice of constructing larger components by aggregating smaller ones (i.e. *granularity mismatch*). Usually large software components have *interfaces that correspond to different aspects of system design* [RX97, p. 32], like configuration, management, operation, monitoring and reliability. These different service categories are used in different domains of interaction. Ran and Xu argue that this separation is often lost at the later stages of design and is rarely used for structuring interfaces, and as a consequence partition component interface descriptions by domains of interaction in which the component may participate. This hides possible interactions between domains from the interface and provides several benefits, namely *separation of concern*, *reuse*, and *controlled propagation of change* (see Figure 2.4 for an overview).

2.4.2 Message Interfaces

In the context of dependable systems the DSoS conceptual model [GIJ+02] defines an interface as a point of interaction between a system and its environment. At the physical level, for instance, an interface can exist as a single line (a serial port) or a set of lines (a parallel interface). An interface can be an output interface or an input interface or both, i.e. a bi-directional interface: An output interface is an interface of a system at which information is produced for the environment of the system. A system without an output interface is meaningless, since it cannot deliver information to its environment and, therefore, has no effect on the environment. An input interface is an interface at which information is consumed from the environment of the system. It is possible to have systems without an input interface (e.g., a clock that produces periodic signals without an explicit input).

In distributed computer systems, the components interact by the exchange of messages across service interfaces to realize the emergent services. A service interface that is provided to link components together is called a LIF. Based on the analysis of the interactions between a component and its environment four different types of component interfaces as depicted in Figure 2.5 can be defined [KS03a]:

The Service Providing Linking Interface (SPLIF): Fundamentally, a component must be a unit of service provision. The service is offered to the component environment across a service providing linking interface (SPLIF). SPLIF is the primary interface of a component.

The Service Requesting Linking Interface (SRLIF): In order to meet its specification, a component may request services from other components. The corresponding interface is the service requesting linking interface (SRLIF). A user of the SPLIF may not be aware that a component requests the services of other components via SRLIFs to achieve its objectives. Thus, the SRLIF s are not visible to the user of a component service.

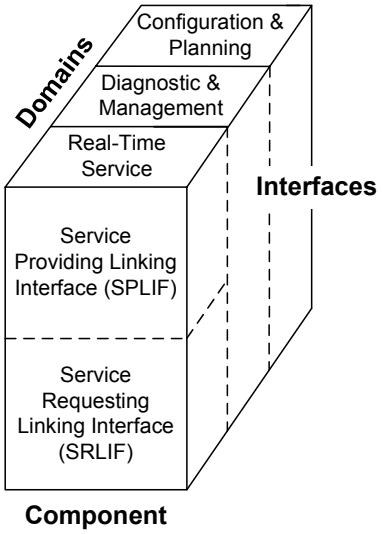

Figure 2.5: Linking Interfaces

The Configuration and Planning (CP) Interface: This is analogous to a global resource manager used to configure – initial and subsequent – components to provide the stipulated services in a specified environment.

The Diagnostic and Management (DM) Interface: The DM interface provides selective access to the (operational) internals of the component for monitoring and diagnostic purposes. Since the DM interface is not exposed during normal component operation, the DM view is not relevant for the user of a component. The DM interface can be used to parameterize a component in order to optimize it for the given task.

Message Interface Specification

The partitioning of the system structure into components with interaction among small and stable linking interfaces serves the purpose of coping with complexity. This design principle of partitioning the system along well-specified interfaces is underpinned by several studies. For example, Lutz stresses the fact that a major source for the occurrence of the critical anomalies (i.e. anomalies leading to subsequent system failures) involve interface specifications, because of an inadequate understanding of the interface [Lut93].

In the context of distributed real-time systems we distinguish between operational and meta-level specification of linking interfaces of components [KS03a]. The operational interface specification includes the syntactic and temporal specification, whereas the meta-level interface specification defines the according semantics.

The *syntactic specification* defines the structure and name of the data elements exchanged via the interface. Thus, the concept of syntax is used to construct structured information from basic information units (e.g., an “integer” consists of 16 bits,

where a bit represents the basic information unit). As an example consider the OMG Interface Definition Language (IDL), which constitutes a key part in the interoperability concept of CORBA [OMG02]. The IDL provides a syntactic description of how a service provided by an object supporting this interface is accessed.

The *temporal specification* determines the temporal sequence of message exchanges. In order to guarantee temporal error detection a priori knowledge about temporal constraints must be available. In case of time-triggered systems, this information about message send and receive instants is existing at design time. In contrast, in many non safety-critical applications the temporal specification includes probabilistic assumptions. Consider for example event-triggered systems, where normally the specification of the inter-arrival times of messages are described by probability distributions [Kle75]. The use of probabilistic specifications complicates a sound diagnostic concept because of the lack of sharp boundaries between correct and incorrect behavior of components.

The *semantic specification* assigns a meaning to each structured information. Depending on the context and deviating understanding of underlying concepts, different conceptual models (i.e. a set of well-defined concepts and their interrelationships [Kop97]) exist. The semantic specification ideally assures that the meaning of the structured information in all involved components is in agreement with the user's intent. A fundamental prerequisite for a complete semantic specification is the existence of an ontology, i.e. *a specification of a representational vocabulary for a shared domain of discourse* [Gru93]. The absence of such an ontology in the application domain leads to a mismatch of the conceptual models and subsequently to imprecise specifications. Furthermore the explanations for design rationales are usually tailored to a particular audience and involve many implicit decisions that complicates a thorough understanding [Eas93].

According to [KS03b] it is impossible to provide a meaningful operational and meta-level LIF specification of a component processing inputs from the natural environment without considering the context-of-use of the component in a particular application environment. Consequently, an understanding of the interface specification is only possible with knowledge from the application domain and the underlying conceptual model (the application designer and component designer need to have the same understanding of concepts, i.e. a common ontology).

As a result many interface specifications lack precision in the semantic domain. With the use of formal methods the introduced ambiguities and fuzziness can be significantly reduced but not removed. Therefore, a complete formalization of the environment would be necessary.

2.5 Dependability

Dependability of a computing system is defined as *the ability to deliver service that can justifiably be trusted* [ALR01].

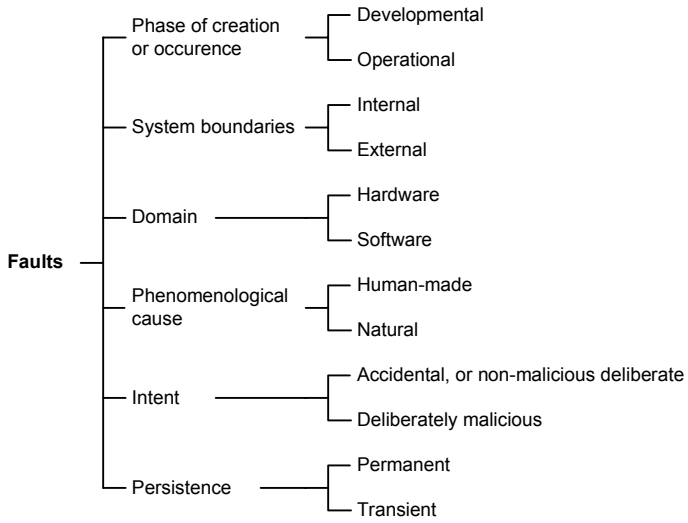

Figure 2.6: Elementary Fault Classes

2.5.1 Fault, Error, Failure

The terminology of fault, error and failure is used according to the definitions found in Laprie [Lap92]. A *failure* is an event that occurs when the delivered service deviates from correct service. An *error* is part of the system state that may cause a subsequent failure. A *fault* is the adjudged or hypothesized cause of an error. As stated in [ALR01] the concept of a fault is introduced to stop recursion. The adjudged cause varies upon the chosen viewpoint (e.g., developer, semiconductor physicist, maintenance engineer). A fault is called active when it produces an error, otherwise it is called dormant.

The elementary fault classes as illustrated in Figure 2.6 are the phase of creation or occurrence, system boundaries, domain, phenomenological cause, intent and persistence. According to their activation reproducibility faults are categorized into solid (hard) or elusive (soft) faults. By combining the elementary fault classes the three combined fault classes design faults, physical faults and interaction faults can be derived.

2.5.2 Fault-Tolerance

Fault-tolerance comprises all methods and techniques intended to preserve the delivery of correct service, i.e. consistent with its specification, in the presence of active faults. Four techniques exists that need to be utilized in order to develop dependable computer systems [Lap92]:

Fault prevention Methods and techniques in order to prevent the occurrence or introduction of faults. This is also known as fault-intolerance [Avi75] or fault avoidance [RLT78].

Fault tolerance Methods and techniques aimed at delivering correct service in spite of faults.

Fault removal Methods and techniques in order to reduce the number or severity of faults.

Fault forecasting Methods and techniques aimed at evaluating and modeling the present number, the future incidence, and the likely consequences of faults.

In [Pol95b, Pol95a] a distinction between application-specific fault-tolerance and systematic fault-tolerance is made. *Application-specific* fault-tolerance subsumes methods that use reasonable checks to detect errors and state estimations for continued operation despite of faults. These reasonable or plausibility checks are based on profound application knowledge and are used to judge about the correct operation of a component. *Systematic fault-tolerance* is based on replication of components, where divergence among replicas is used as a criterion for fault-detection. A key advantage of systemic fault-tolerance is the fact, that no application knowledge or assumptions about the controlled object is needed.

In [Bau01] the systemic transparent fault-tolerance techniques used in the Time-Triggered Architecture (TTA) are described. Here, fault-tolerance mechanisms are handled at system-level and validated once and for all, relieving application designers from increasing the inherent application complexity by including systemic mechanisms.

2.5.3 Fault and Error Containment

In any fault-tolerant architecture it is important to distinguish clearly between fault containment and error containment [Kop03]. Fault containment is concerned with limiting the immediate impact of a single fault to a defined region, while error containment tries to avoid the propagation of the consequences of a fault, the error. It must be avoided that an error in one fault-containment region propagates into another fault-containment region that has not been directly affected by the original fault.

The 10^{-9} Challenge

Emerging X-by-wire applications require ultra-high dependability in the order of 10^{-9} failures/h (115.000 years) or lower. Today's technology cannot support the manufacturing of electronic devices with failure rates low enough to meet the reliability requirements. Thus the reliability of an ultra-dependable system must be higher than the reliability of each of its components. This can only be achieved by utilizing fault-tolerant strategies that enable the continued operation of the system in the presence of component failures [BCV91]. Since systems can only be tested to a dependability in the order of 10^{-4} failures/h a combination of experimental

evidence and formal reasoning using a reliability model is needed to construct the safety argument. The safety argument is a documented body of evidence in order to convince experts in the field that the provided system as a whole is safe to deploy in a given environment [BB98].

The justification for building ultra-reliable systems from replicated resources rests on an assumption of failure independence among redundant units. For this reason the independence of Fault Containment Regions (FCRs) (i.e. subsystems that share one or more common resources and may be affected by a single fault) is of critical importance. Thus any dependence of FCR failures must be reflected in the dependability model. Independence of FCRs can be compromised by

- Shared physical resources (hardware, power supply, time base, etc.)
- External faults (EMI, heat, shock, spatial proximity)
- Design [KBJ00]
- Flow of erroneous messages

If complex systems constructed from components with interdependencies are modeled, the reliability model can become extremely complex and the analysis intractable [BCV91].

Fault Containment

The notion of a FCR is introduced in order to delimit the immediate impact of a single fault to a defined subsystem of the overall system [LH94]. A FCR is defined as the set of subsystems that share one or more common resources and may be affected by a single fault. Since the immediate consequences of a fault in any one of the shared resources in an FCR may impact all subsystems of the FCR, the subsystems of an FCR cannot be considered to be independent of each other [KJ00]. The following shared resources that can be impacted by a fault are considered:

- Computing Hardware
- Power Supply
- Timing Source
- Clock Synchronization Service
- Physical Space

For example, if two subsystems depend on a single timing source, e.g., a single oscillator or a single clock synchronization algorithm, then these two subsystems are not considered to be independent and therefore belong to the same FCR. Since this definition of independence allows that two FCRs can share the same design, e.g., the same software, design faults in the software or the hardware are not part of this fault-model.

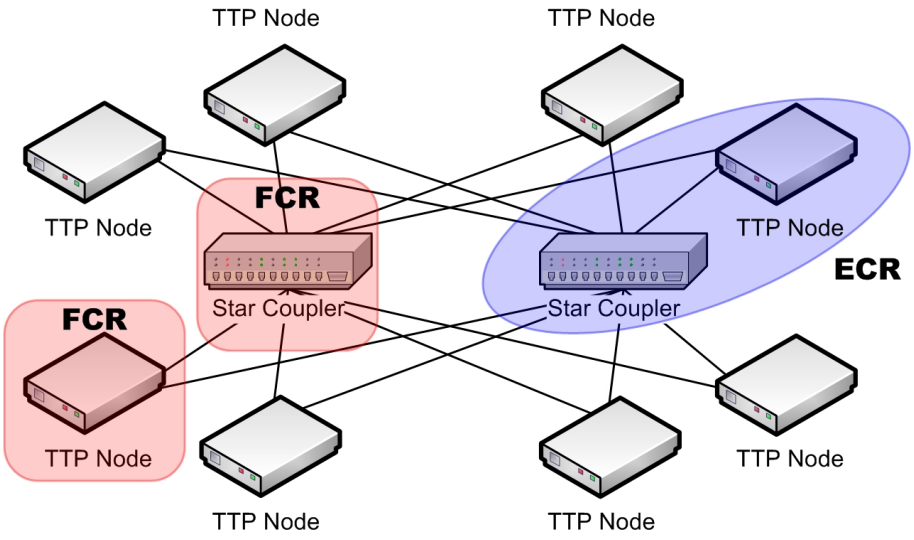

Figure 2.7: FCRs and ECRs in the Time-Triggered Architecture

Error Containment

An error that is caused by a fault in the sending FCR can propagate to another FCR via a message failure, i.e. a sent message that deviates from the specification. A message failure can be a message value failure or a message timing failure [Cri91]. A message value failure implies that a message is either invalid or that the data structure contained in a valid message is semantically incorrect. A message timing failure implies that the message send instant or the message receive instant are not in agreement with the specification. In order to avoid error propagation by way of a sent message error-detection mechanisms that are in different FCRs than the message sender are needed, i.e. an Error Containment Region (ECR) requires at least two independent FCRs. Otherwise, the error detection mechanism may be impacted by the same fault that caused the message failure.

For example in the TTA [KB03], timing failure detection is performed by a central guardian, while value failure detection is in the responsibility of the host computer (e.g., using a Triple Modular Redundancy (TMR) configuration). Since fault injection experiments have shown that for ultra-dependable applications restrictions concerning the failure modes of ECUs are unjustified [ASBT03], independent central guardians [BKS03] are needed for achieving fault isolation in case of arbitrary ECU failure modes. In the TTA each central guardian is an autonomous unit that has a priori knowledge of all intended message send and receive instants [KBS05]. Each of the two replicated channels has its own independent guardian. Figure 2.7 depicts the FCRs and ECRs in the TTA.

Chapter 3

Related Work

This chapter presents related work in the context of diagnosis and maintenance. At first we elaborate on error and anomaly detection followed by a summary of today's prevalent maintenance strategies, namely time-based and condition-based maintenance. Furthermore, an overview on automotive and avionic infrastructures for diagnosis and maintenance is given. In particular, today's automotive electronic architectures are discussed as well as prevalent diagnostic strategies including the OSEK/VDX network management approach. In the context of avionics representative design guide lines, such as ARINC 624, for on-board maintenance systems are presented. Moreover, the diagnostic infrastructure of the Boeing 777 is briefly introduced. In order to understand the relevant factors for the increasing ratio of fault-not-found phenomena industry is currently facing, the changes in the design of IC's and accompanying impacts with respect to maintenance are discussed. Examples from automotive and avionic industry illustrate the need for an accurate diagnostic solution to reduce the number of avoidable component replacements. Furthermore, the Heinrich pyramid and related models are presented. The chapter is concluded with a review on existing analysis techniques, such as threshold-based techniques, probabilistic networks and model-based diagnostic algorithms.

3.1 Error and Anomaly Detection

3.1.1 Error Detection using Assertions

Assertions are boolean expressions that test the state of an executing program. Typically assertions have a form similar to

`if not` ASSERTION `then` ERROR

where ASSERTION is a predicate on the program state and ERROR indicates that an error condition is triggered.

According to [Hoa00] assertions were first proposed by Turing and rediscovered by Floyd in order to assign meanings to programs [Flo67]. This lead consequently to the use of assertion for formal reasoning about the correctness of sequential programs [Hoa69]. Assertions provide a logical basis for proofs of the properties of a program. Hoare introduced the so-called Hoare triple $P\{Q\}R$ to make general assertions about the values of relevant state variables. The precondition P expresses the values before, while the postcondition R expresses the values after the execution of program Q. The final assertion R implies the desired property. More information on reasoning for sequential programs using the Hoare logic can be found in [Apt81].

In addition to verification, assertions are also a powerful and acknowledged mechanism to improve the reliability of software [AAS79]. So-called *executable assertions* are embedded into the code and provide automatic runtime detection of errors [Hoa00, MN88, Ros92].

The effectiveness of these embedded assertions depends heavily on the context of use [Hil99]. Though assertions are a very useful technique in detecting and diagnosing problems, the execution of software assertions decreases efficiency [VM94]. In addition blind checking (i.e. including as many checks as possible into the software) introduces more software and thus more possible sources of software faults. For this reason a more systematic approach, namely the design-by-contract software engineering technique for object-oriented design, is proposed in [Mey92].

3.1.2 Anomaly Detection

The phenomenon of system anomalies is subject to intensive research in many domains. This section gives a brief overview about prevalent definitions of the concept of anomaly.

Brotherton et al. [BJ01, p. 3113] define anomalies in the context of aerospace systems as *off-nominal operations that have never been anticipated nor ever encountered before.*

In the context of embedded systems Maxion et al. [MT02] define the task of an anomaly detector to quantitatively decide on the distance between normal and abnormal behavior of a component based on a similarity or distance metric. The actual interpretation of the reference behavior and deployed similarity metric are usually application specific.

Anomaly detection is often based on a specific application model (e.g., jet engine), in order to compute the difference between variable values and the model-estimated outputs. The results are used to determine whether these differences indicate an anomaly compared to nominal test case [JW02].

During spacecraft missions each occurring system anomaly is reported for further analysis and assessment. A so-called ISA (Incident/Surprise/Anomaly) report is filled out by the operator at mission control at the time of occurrence. Later this report is extended by the analysis and concluded by the description of a possible

corrective action. In contrast to a defect report, an ISA is written whenever the *behavior of the system differs from the expected (i.e. required) behavior*. The importance of this ISA lies in the provision of valuable information to the requirements engineer by capturing the gaps between the specified and implemented requirements and the user's expectation [LM01].

3.1.3 Comparative Evaluation

Assertions are a powerful and accepted mechanism facilitating in the detection of value errors. In particular, assertions are used in software engineering to ensure correctness and safety requirements. Recent studies revealed that the detection coverage of executable assertions is fairly high [Hil00].

However, a prerequisite for the deployment of bivalent assertions is the ability to indisputably demarcate between correct and incorrect system states at the time of occurrence. System anomalies can be described in terms of a deviation from the expected nominal system behavior. However, the construction of a similarity metric is a complex task and is based on profound knowledge about the application. In many application domains, it is difficult to devise metrics and thresholds. For instance, an unusual code path due to an unanticipated combination of circumstances can cause unexpected behavior [LM01]. For this reason it is necessary to devise a classification scheme that refrains from the traditional correct/incorrect classification. This diagnostic strategy allows to cope with system anomalies that cannot be judged as being correct or incorrect at the time of occurrence.

3.2 Preventive vs. Corrective Maintenance

Today two well established maintenance approaches can be found [PP01]:

Time-Based Maintenance (TBM): This is the predominant maintenance type in the automotive industry. Time-Based Maintenance (TBM) is carried out in regular time intervals suggested by the equipment manufacturer. After a specified time period or milage (e.g., 2 years or 50.000 km), the car owner has to bring his car to the service station to check for problems or wearout symptoms (e.g., motor oil, brake pads). The main disadvantage of this maintenance method is either too early or too late maintenance.

Condition-Based Maintenance (CBM): CBM optimizes the intervals between preventive maintenance checks by monitoring the condition of the equipment. This method has two advantages. Firstly, savings in maintenance can be achieved, if the maintenance intervals for CBM are longer compared to TBM. Secondly, if the maintenance intervals are shorter, component failures and associated consequences can be avoided. Thus not only costs are reduced but also the confidence of the customer in its car is increased. One major drawback of

CBM is the additional complexity and functionality needed for the condition assessment process.

TBM is increasingly being replaced by CBM, to reduce costs and to improve reliability and system performance [TS01]. Originally introduced in the avionics domain, this new paradigm is more and more accepted in the automotive industry. Besides the reduction of cost of ownership (service only what is needed) the possibility of collecting accurate field data (*engineering feedback*) is one of the major benefits from this maintenance approach [Dei02].

In [GF03] methods for forecasting of short-term part demand are investigated. Such spare parts, though low in demand, are often critical in operation and the unavailability can lead to expensive down time. For example it is estimated that an aircraft operator can incur more than 50.000 USD for each hour if a plane is on the ground. In the avionic domain three categories of component maintenance strategies can be identified [GF03]: *hard-time*, *on-condition*, and *condition-monitoring*. While hard-time can be compared to the previously introduced TBM scheme, on-condition maintenance requires that the part under inspection to be periodically checked against some appropriate physical standard to determine whether it can continue in service. On-condition maintenance is a preventive maintenance technique. Finally, condition-monitoring requires the continuous eventuation of the respective part to judge about the need for corrective procedures.

However, CBM has also some downsides like increased complexity and the need to continuously monitor and assess the state of the components of the system. This is a challenging task in the design and engineering process of new components. According to [WCRV00] the implementation of CBM requires

- Knowledge about component failures, deterioration, and criticality
- Suitable indicators for the status and degradation of components
- Diagnostic tools to measure these indicators
- Assessment tools to reliably interpret measurements

For example in machinery vibration, thermal, and lubricant analysis are good indicators for possible defective conditions [Sta97]. In [PK03] a case study for wear and debris analysis is presented.

In order to adopt CBM for electronic systems suitable indicators for degradation or wearout must be identified and analyzed to detect deviations from sound operation. Consider for instance the wearout of brake pads in an automotive braking system [Rei03]. On a disc braking system, once the driver pushes the braking pedal, the fluid from the master cylinder is forced into a caliper where it presses against a piston. As a result the piston squeezes two brake pads against the rotor forcing the car to decelerate. Condition-based maintenance can here be used to judge the condition of the braking pads by measuring the distance the brake pads need to overcome

before having contact with the rotor. Similar to this example, suitable indicators for electronic wearout need to be identified. A promising indication of premature electronic wearout is the increase in the transient failure rate of a component. In the Section 3.5, this important issue is discussed in detail.

3.3 Automotive Diagnosis and Maintenance

All modern cars are equipped with on-board diagnosis systems. However, the development of effective diagnostic systems stayed behind the recent increase of electronic systems in modern cars. One reason for the diagnostic deficiencies of modern OBD systems is the fact that diagnosis is often treated as add-on to communication systems rather than an integral part of the architecture [SSW00]. Consequently the problem of the identification of faulty ECUs is one of the predominant challenges that needs to be solved.

Today's economic pressure in the automotive industry forces the introduction of new car models in decreasing intervals. In recent years the vehicle development cycle was reduced from four to two years to cope with market demands [Bar01]. As a consequence of the reduced vehicle deployment cycle the technical documentation is insufficient to resolve a significant number of vehicle problems (impact in the first 2 years). These technological and business realities underpin the need for effective diagnosis methods because it takes six months on average for the service technicians at the garages to gain experience with a new car (often due to insufficient technical documentation). In addition, emerging X-by-wire solutions will have a lasting effect on the mechanics work, since computer diagnostic will become a standard part of the job [LH02, Bre01].

Since a mechanic at a service station is no specialist in automobile electronics, the diagnostic system of the car must provide all necessary information, that allow maintenance of faulty components. For this reason it must be possible in modern automotive electronic architectures to track an entry in a breakdown log back to its source. If this is not possible, as a consequence, fully operational units will be replaced by mistake.

The primary goal of every car manufacturer is overall consumer satisfaction. A car worth 50.000 Euro that is disabled by the malfunction of an undetected 50 Cent connector or ECU is thus not tolerable. An automobile manufacturer will only be able to bind its customers, if the cost of ownership and the time to repair can be kept low.

3.3.1 Automotive Infrastructure

To give an impression of the complexity and the amount of electronics in today's luxury cars take for example the electronic infrastructure of a luxury car depicted in Figure 3.1. The distributed ECUs of each federated cluster of the car are interconnected via communication networks with different protocols (e.g., Controller

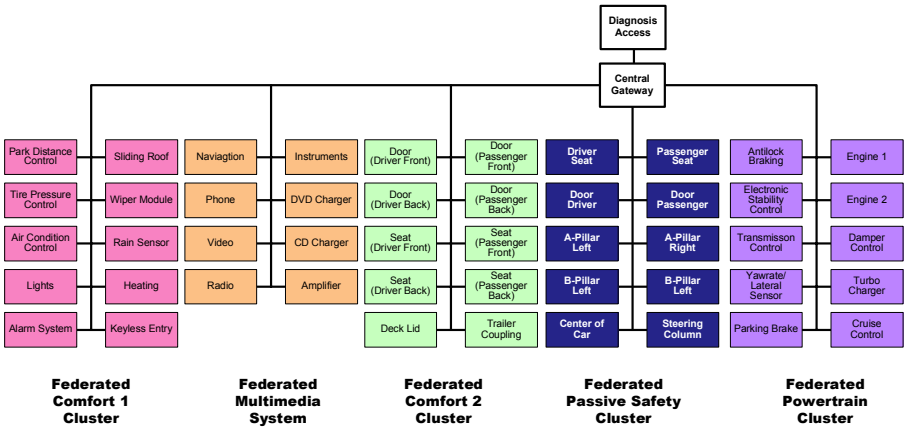

Figure 3.1: The Electronic Infrastructure of a Luxury Car [Dei02]

Area Network (CAN) [Bos91], Local Interconnect Network (LIN) [Fle03]), physical layers, bandwidths (10 kbps–500 kbps), and dependability requirements. Multiple federated clusters are connected via a central gateway allowing data exchange and access to the OBD systems of each ECU. The comfort clusters as well as the powertrain cluster are typically implemented via the CAN protocol. The multimedia cluster is frequently based on a protocol with support for streaming audio and video (e.g., MOST [MOS02]), while the passive safety clusters either use CAN or vendor-specific communication protocols such as byteflight [BG00]. Each of these clusters consists of components designed according to the "1 Function – 1 ECU" principle in order to simplify system integration and ensure intellectual property protection. Consequently, additional ECUs need to be added to the clusters in order to improve the functionality of the car. For instance, the Volkswagen Phaeton can have up to 61 ECUs, the BMW 7 up to 75 ECUs depending on the customers requested extra equipment [HR02, Dei02]. However, this trend of increasing the number of ECUs is coming to its limits, because systems are becoming too complex and too costly with the current practice of having each ECU dedicated to a single function. With an average cost of 30-50 Euros per ECU this high number of ECUs bears significant potential for cost reduction [POT+05].

System Integration

During system integration significant efforts are caused by unanticipated interactions between subsystems provided by different vendors. The sharing of communication resources in today's cars across different subsystems (e.g., systems based on the CAN protocol) makes it hard to fully test the functionality of a subsystem in isolation as it will be integrated in the car. As a consequence, there is the need for a comprehensive integration test by the car manufacturer to determine possible mutual interference of subsystems. In contrast, a system architecture with rigorous operational interface specification [KS03a] and error containment can avoid the introduction of mutual interference during system integration. Such a temporally composable ar-

chitecture [KO02] exhibits the benefit of dramatically decreasing integration costs, because the validity of test certificates from suppliers is not invalidated during system integration.

Complexity Control

Each subsystem (e.g., engine control, brake assistant) possesses a functional complexity that is inherent to the application. The functional complexity of a subsystem when implemented on a target system is dramatically increased in case the architecture does not prevent unintended architecture-induced side effects at the communication system. Since federated systems employ a dedicated computer system for each subsystem, the complexity of the system is lower compared to the integrated systems approach. The absence of interactions and dependencies between subsystems reduces the cognitive complexity to a manageable level. In today's cars we do not find a totally federated architecture nor an integrated one. In fact, the economic pressure in the automotive industry requires system designers to utilize the available communication resources for more than one subsystem without protecting the resources from mutual interference. For a deeper understanding consider an exemplary scenario with two subsystems. If the two subsystems share a common CAN bus, then both subsystems must be analyzed and understood in order to reason about the correct behavior of any of the two subsystems. Since the message transmissions of one subsystem can delay message transmission of the other subsystem, arguments concerning the correct temporal behavior must be based on an analysis of both subsystems. In a totally federated system, on the other hand, unintended side effects are ruled out, because the two subsystems are assigned to separate computer systems.

3.3.2 Diagnostic Infrastructure

Figure 3.2 shows the diagnostic infrastructure of today. Each ECU deployed in a car typically has a diagnostic subsystem that analyzes the functionality of the constituting parts (e.g., via Built-In Self Test (BIST)) or performs application specific plausibility checks, i.e. assertions, to detect errors.

Once the OBD system of the car detects a violation of the specification of an ECU, a breakdown log entry is written, and in case of a high severity, the driver is informed via the Malfunction Indicator Light (MIL). In case of an error, current diagnostic systems provide a so called *freeze frame* function, that records the condition of the vehicle when a failure occurs. The freeze frame provides important information for the failure cause analysis. The breakdown-log typically stores data on the type of fault, the state of the system, the priority, the environmental conditions, a timestamp, and information on the milage of the car. Depending on the type of inspection (e.g., garage, factory inspection, development) different parts of the breakdown log entry are analyzed.

In maintenance mode the ECUs are accessed using dedicated protocols like ISO-9141, J1850 or the CAN based KWP 2000 [Wal02]. At the service station the me-

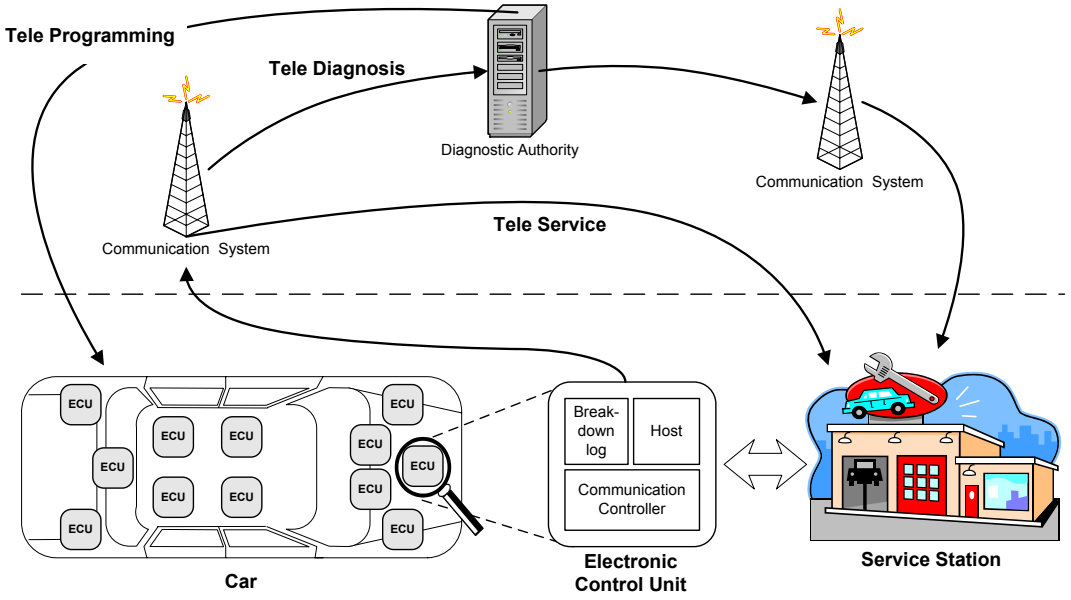

Figure 3.2: Automotive Diagnostic Infrastructure

chanic uses a diagnostic testing device (e.g., VAG tester) to receive information about pending problems. Since most mechanic are no specialists in automotive electronics, the service technician depends on the accuracy of the diagnostic information provided by the OBD. At the garage only a specific testing device code (including a short human readable textual description) and a Diagnostics Trouble Code (DTC) are read and interpreted. A DTC provides additional background information on the failure, for example an 8 bit value describing prevalent faults. Based on this information the mechanic must be able to decide which part of the system caused the failure and needs to be replaced to restore full functionality.

The dashed line in Figure 3.2 indicates the cut between current available technology and advanced services that are subject of current research. Future trends like tele-diagnosis, tele-service and tele-programming offer new possibilities in the collection of diagnostic information and customer service [Dei02]. The ultimate goal is to shorten the delay between the occurrence of a failure and the definition of corrective action.

Tele-service is used to automatically inform the dealer about wearout and tear of the car via a telephone network (GSM, UMTS). The service call is then inspected by the service advisor who supervises the needed services, checks the availability of the parts to be replaced and makes the necessary appointments.

The vision of tele-diagnosis is to send detailed diagnostic information of the car to a diagnostic authority that processes and analyzes the collected data. Thus not only malfunctions of components are reported (like OBD) but also valuable information of data from vehicles on the road can be collected and analyzed in order to enhance diagnostic procedures [MRS+02]. However, data privacy issues have to be clarified by the legislator.

Tele-programming is supposed to allow the programming of the car electronics either to correct software faults or to extend the functionality of the car. For more detailed information on telematics technology trends refer to [JM02] and [Lan02].

3.3.3 Role of Diagnosis

Three related diagnostic tasks can be identified in advanced automotive system, namely fault detection, identification of faulty Fault Containment Regions (FCRs) and fault identification within a FCR, which will be discussed in the following.

Failure Detection

Failure detection is the identification of a deviation of the provided service from the intended specification of a component of the system. Failure detection is the minimal function that any OBD system of a vehicle must perform [AM02].

One of the most important parameters in the design of failure detection is the sensitivity of the analysis algorithms [OM02]. By designing the algorithms too sensitive the likelihood of faulty classification of operational components will increase significantly. Unnecessary MIL activations will have a lasting effect on the user's trust into his car [Ber02]. For this reason setting the scope of the analysis parameters is a tradeoff between the frequency of faulty detection and detection of faulty nodes.

During the factory inspection process the situation is different. The required level of diagnosis is far more sensitive than the diagnostics functions adopted for market conditions. For this reason it is necessary to develop diagnosis systems that can change the malfunction detection sensitivity by means of a diagnostic scan tool. According to [OM02] this is an effective weapon in the phenomenon investigation for defects with minimal repeatability.

Identification of faulty FCRs

This service of a diagnostic system provides a determination of the exact location of the fault within the system, i.e. which FCR deviates from its intended function.

Barkai [Bar01] provides some interesting numbers that underpin the current problems of diagnostic services in automotive communication systems. He states that in more than 20% of MIL activations, the OBD did not provide sufficient data to identify a root cause and were dismissed as No Fault Found (NFF). Furthermore Original Equipment Manufacturer (OEM) studies show alarmingly high rates of incorrect initial diagnosis of electrical problems, in some cases exceeding 50%.

This numbers point out the lack of current communication systems to provide assistance in the fault isolation process. As a consequence the car repairman chooses the simplest solution and changes all components that can be responsible for the

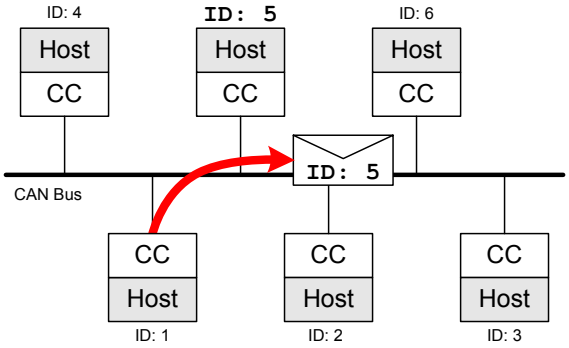

Figure 3.3: Diagnostic Deficiencies of CAN: Masquerading

malfunction [MRS+02]. This procedure of throwing away operational components dramatically increase the costs of a car either for the user (costs of ownership) or the manufacturer (warranty repair costs).

To illustrate one of the diagnostic deficiencies of the widely used CAN [Bos91] communication protocol, consider the following example. Due to a transient (or permanent) fault a node forges the ID of a message. This phenomenon, also called masquerading, is not prevented by CAN protocol mechanisms. Masquerading is defined as the *sending or receiving of messages using the identity of another principal without authority* [CDK94, p. 480]. Consequently, even an external observer cannot identify the faulty sender. Figure 3.3 depicts this scenario. The faulty node with ID 1 sends a message to all other nodes and it pretends to be node with ID 5. Such a forged message ID can have a significant impact on the application. Once the failure is detected and an entry is written into the breakdown log the service technician at the garage will most likely change the node with ID 5 that caused the receiving node to fail due to an incorrect message. Hence the faulty node remains undetected and unchanged.

Fault Identification within the FCR

After the faulty FCR within the system has been identified, the fault identification process classifies the fault and determines the root cause. Fault identification is mostly an offline activity due to high complexity and effort required. In automotive applications fault identification is typically applied to returned parts or for warranty analysis [AM02] by the OEM. Studies of the ECUs used in automotive applications underpin the so-called Pareto-principle, i.e. a phenomenon that can have many theoretical causes has in reality only a few [PMH98]. Recent studies show that the components with the highest failure rate are Printed Circuit Boards (PCBs) and micro-controllers followed by analog ICs and ASICs (the higher the integration, the more likely the component is subject to fail). Resistors, transistors and diodes have the lowest failure rates. More detailed information on ECU failures and component failures can be found in [WWS99].

3.3.4 OSEK/VDX

OSEK (Open Systems and the Corresponding Interfaces for Automotive Electronic) is a joint project in the German automotive industry aiming at an industry standard for distributed control systems in cars. When the French automotive industry joined the consortium, the VDX-approach (Vehicle Distributed eXecutive) has been included into the standard. The OSEK/VDX architecture comprises:

- Communication
- Network Management
- Operating System
- OSEKtime Operating System
- OSEKtime Communication (FTCOM)

In the remainder of this section we will focus on the Network Management (NM) [OSE03]. The OSEK Network Management defines a set of services for the network management of the nodes deployed in a distributed control network within a vehicle. Among these services are interfaces to the application, algorithms for node monitoring, OSEK internal interfaces (e.g., with the OSEK Communication standard), an algorithm for transition into sleep mode and the Network Management protocol data unit.

Since the scope of this thesis lies on diagnosis we will only consider parts of the standard relevant to this topic. Node monitoring in the scope of OSEK/VDX is used to inform the application about the status of the nodes on the network. The NM offers two alternative mechanisms for network monitoring that will be discussed next.

Direct Network Management Concept

OSEK supports direct node monitoring by dedicated NM communication. In the OSEK scheme every node of the network actively monitors each other node. Based on the proposed diagnostic algorithm, every node broadcasts periodically diagnostic NM messages as a life sign in a logical ring topology. The communication sequence in this logical ring is independent of the actual in-vehicle network structure. Therefore each node is assigned a logical successor. OSEK distinguishes two types of messages. Alive messages are used to register new senders to the logical ring. A ring message is responsible for the synchronized running of the logical ring. It will be passed from one node to another (successor) node. During operation a time-out on a ring message will be interpreted as a node failure. A node is also classified as faulty, if the NM messages sent by the node indicates an erroneous state (i.e. the node declares itself as faulty). However, the identification of the exact cause for a node failure is not part of the NM. The implementation of this diagnosis scheme requires several timing values

Parameter	Definition	Validity
T_{Typ}	Typical interval between two ring messages on the bus	global
T_{Max}	Maximum time interval between two ring messages	global
T_{Tx}	Delay to repeat the transmission request	local

Table 3.1: Network Management Timing Parameters

to be respected as listed in Table 3.1. Based on the values of monitoring counters (with a specific threshold), the nodes of the system are considered operational or not.

Indirect Network Management

In case direct monitoring cannot be applied, OSEK introduces indirect monitoring mechanisms. Indirect network management uses monitoring of periodic application messages to determine the health status of the control units connected to the network. This monitoring scheme makes no use of dedicated NM messages. However, the use of this technique is limited to nodes that periodically send messages in the course of normal operation.

In this case, a node emitting such a periodical message is monitored by one or more other nodes receiving that message. Nodes whose normal functionality is limited to receiving must send a dedicated periodic message in order to be monitored.

In the optional extended configuration management the status of the participating nodes are determined by the use of counters with a specific threshold indicating the malfunction of a node.

3.4 Avionic Diagnosis and Maintenance

This section gives a short overview on the maintenance strategies and implementations of on-board maintenance solutions onboard commercial aircrafts, in particular on the basis of the Boeing 777 by-wire plane [Yeh98].

3.4.1 On-bard Maintenance System

In order to reduce "best guess, shotgun maintenance" [SKSS94] in avionics, aircraft manufacturers provide on-board maintenance solutions following the design guidelines of documents like ARINC 624 [Aer93]. According to [Aer96] approximately 50% of all equipment removals are reported as NFF, i.e. *electronic equipment removed from an aircraft during maintenance troubleshooting, which, when returned to the manufacturer, is tested and found to work correctly.*

ARINC 624 As for any diagnosis and maintenance system, ARINC 624 emphasizes, that *unreliable diagnosis is worse than no diagnosis*. ARINC 624 provides design guidelines for an Onboard-Maintenance System (OMS) for state-of-the-art aircrafts. Thereby, the main objectives of an OMS is to serve as the tool for consolidation and correlation of all Built-In Test Equipment (BITE) results for centralized access and display:

- Cost-effective and user friendly means of airplane maintenance
- Reduction of shotgun maintenance
- Simplification of maintenance procedures
- Elimination/reduction of ground support equipment
- Provision of an Airplane Condition Monitoring System (ACMS) for the monitoring of performance trends

A key aspect of ARINC 624 is the reduction of efforts required by maintenance personal by automating ground tests and minimizing the need for operator interactions. To ease maintenance, the OMS should also provide an overview about the installed Line Replaceable Units (LRUs) onboard the aircraft. Furthermore, the OMS allows ground personal to correlate reported system anomalies by the aircraft crew with BITE records in the system. For this reason, for each entry in the database contextual information is stored, such as

- Failure indication or flight deck effect, if any
- Flight phase and flight leg
- Time and date
- Flight number and city pair or route number
- Airplane identification
- Airplane flight parameters to support the pilot report and/or troubleshooting, e.g., altitude, airspeed etc.

Since airlines typically require large amounts of maintenance documentation, the OMS should provide an interface to allow accessing an electronic library system. This way the maintenance engineer has access to relevant information for maintenance activities for each deployed LRU onboard an aircraft.

ACMS As part of the OMS, the Airplane Condition Monitoring System (ACMS) monitors and records selected airplane data related to aircraft maintenance, performance, and troubleshooting. The main goal of the ACMS is the detection and analysis of potential malfunctions (cf. CBM) in order to allow timely maintenance actions resulting in quick turn-around of the aircraft.

The ACMS is required to collect data over a specified period of time referenced to a specific event. This way trend reports can be generated for either engineering feedback or anomaly analysis.

Another fundamental function of the ACMS is also providing access to the current health status of fault-tolerant subsystems (e.g., the fly-by-wire system) of the aircraft. This way the redundancy status can be assessed by the service technicians, and maintenance activities scheduled before the minimum redundancy level is reached.

Scrubbing Since fly-by-wire applications require ultra-high dependability fault-tolerant strategies need to be utilized that enable the continued operation of the system in the presence of component failures. So-called "scrubbing-techniques"

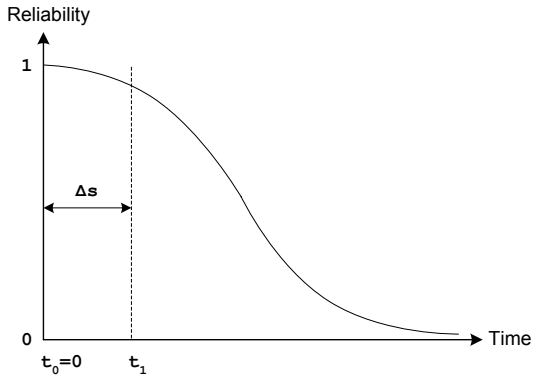

Figure 3.4: Scrubbing Techniques

are used to validate the functionality of fault-tolerance mechanisms in specified time intervals. Figure 3.4 illustrates the basic concept of this approach. The reliability of electronics components decreases with time. At time t_0 the component is assumed to be fully functional. At time t_1 the proper function of fault-tolerance mechanisms is validated again and therefore the component is supposed to have the same reliability as at instant t_0. The likelihood of undetected components that are stuck *at-good-failure*, i.e. the component works as specified as long as no failure occurs, depends on the time interval Δs. Scrubbing is closely related to CBM techniques. However, the system is tested in a special maintenance mode and never in operational mode. Intentionally generated faults are used to examine the specified behavior of all components in the case of a malfunction.

Case Study - The OMS for the Boeing 777 The Central Maintenance Computer (CMC) as part of the Boeing 777 OMS assists in the analysis of flight deck effects and crew complaints from the arriving flight, and diagnoses the reasons behind these symptoms [Ram92]. A key mechanism is the identification of the responsible Line Replaceable Module (LRM) causing the system malfunction. For electronic equipment it is no longer possible to perform simple visual inspection. Thus, it is

necessary to assist the service technician in order to prevent shot-gun maintenance, i.e. the replacing of LRM and hoping that the fault is eliminated. The 777 OMS heavily exploits BITE, on the one hand to eliminate the problem of equipment removals in order to gain access, and on the other hand to allow correlation of electronic failures with flight deck effects. A key functionality of the CMC is to suppress secondary symptoms that may mislead the analysis process. For more information on the user interface of the CMC refer to [VB94]. The CMC analysis is based on a model-based algorithm which encodes the cause-effect relationship. The model defines fault conditions, associated symptoms and repair actions for each LRU on the aircraft [Fel94].

3.4.2 ARINC 653

The ARINC 653 specification [Aer03] as part of the Integrated Modular Avionics (IMA) [Aer91] defines a general-purpose APplication EXecutive (APEX) interface between the operating system of an avionics computer resource and the application software.

Health Monitor

ARINC 653 proposes so-called Health Monitors (HMs) in order to monitor and report hardware and software faults (e.g., application and operating system). The main purpose of a HM is fault isolation and prevention of error propagation. The HM are contained within the following software elements:

Operating System Typically, HM are functions of the operating system, where a HM predefined configuration table defines the appropriate response to exhibited failures.

Application Partitions In case faults are system specific and determined from the logic or calculation, application partitions are used to pass either the detected faults to the operating system or to an appropriate system partition.

System Partitions These partitions can be used by the system integrators as error handlers. The advantage of this approach is that responses to detected faults do not have to be based purely on a look up table and the implementation is outside the operating system.

Error Detection

In conformance with the specification, errors are detected by several elements:

Hardware Typical hardware errors are memory protection violations, overflows, zero divide, timer interrupts or I/O erros.

Core Software At the core software level ARINC distinguishes between configuration faults and deadline violations.

Application Application specific errors comprise sensor failures or discrepancies in multiple redundant outputs.

However, the exact list of detected errors and the location of the error detection mechanisms is implementation specific.

3.5 Setting the Focus on Transients

The increasing rate of transient failures in electronic systems implies consequences to system designers and manufacturers. The so-called *Trouble Not Identified (TNI) phenomenon* (also known as Cannot Duplicate (CND) or No Fault Found (NFF) problem) is probably the most discussed maintenance problem currently affecting both avionics and automotive industry. The following section is devoted to identify reasons for this increase of transients in electronic systems and analyzes the complex interrelationships causing TNI and related phenomena.

3.5.1 The Trouble Not Identified Phenomenon and its Implications

The TNI phenomenon is characterized by the fact that the source of a transient system malfunction cannot be easily identified. Today many used architectures are providing limited and ineffective diagnostic services not allowing the detection of any single anomaly within the system. Due to insufficient fault isolation and limited control of error propagation the identification of faulty components becomes often intractable. Thus, the classification of the experienced failure modes becomes a very difficult, if not impossible task. However, this determination whether the fault source originates from internal or external disturbances is the first step to tackle the TNI phenomenon. To illustrate the difficulties of the identification of such faults consider the following example taken from [TAP02, p. 642]:

> *One aspect of the intermittent problem was that when two components are soldered together and a crack develops in the solder connection, for whatever reason, and then the unit experiences expansion and contraction due to changes in temperatures, the connection may be closed at some times and open at other times.*

This example indicates that these transients can result due to complex environmental condition that cannot be easily reproduced at the time of maintenance.

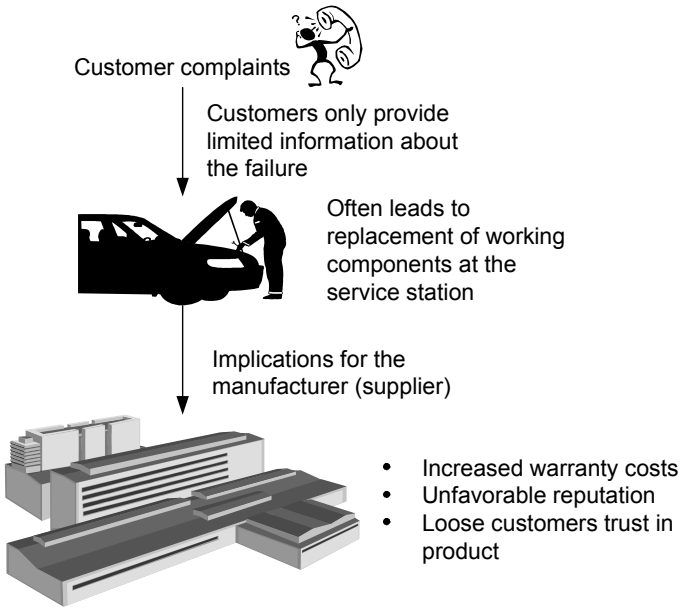

Figure 3.5: The TNI Phenomenon

Automotive Domain

The TNI phenomenon is an increasing problem in automotive electronics causing major economic implications. As depicted in Figure 3.5, once a customer is informed by the OBD system of the car about a malfunction (e.g., by the illumination of a MIL), he informs the car dealer about the failure. However, not all transient failures will be recognized by the customer. These undetected failures impose even more a serious threat to the safety of the system. Usually the mechanics at the service station get limited and unprecise information about the location, symptoms and possible failure cause [TAP02]. This lack of information of why the TNI occurred often results in unnecessary replacements of working components [MRS+02, TE01, Bar01]. Even worse, in many cases the component causing the anomaly in the systems remains unchanged.

Example 1: Ford Ignition Modules In [TAP02] a case study from a Ford ignition module is presented. In 1984 on average 40% of the ignition modules would fail within five years or 50000 miles. The main problem was the fact that many of the modules did not exhibit a failure mode when they were returned. That is, the electronic modules were removed from the car, but passed the tests at the manufacturer. However, Ford identified several failure mechanisms that could cause intermittent faults, including substrate cracks, cracked solder, cracked resistors, shorted leadframes, leadframes unsoldered from housing and others.

The most interesting discovery of the Ford analysis was the lack of feedback from the experienced failure modes, i.e. the lack of support from the used architecture to identify faulty components and to provide field data to the engineers.

Another problem is the identification of the fault source by the module manufacturer (warranty analysis). Visual inspection exposed to be often inappropriate, because microscopic failures can be difficult, if not impossible, to detect, especially if the location of the failure is unknown. As investigated by Ford, it is likely to occur that a module could pass the tests but fail in the field [TAP02].

Example 2: Cruise Control Modules A recent study about failure modes of cruise control units revealed that the intermittent nature of the majority of component failures makes it difficult for the vehicle engineers to identify the fault source [KHTP99]. In fact, more than 96% of the failed cruise control modules removed from the vehicles passed bench tests. The inability to replicate real-world conditions in order to expose the fault during laboratory tests is a prevalent problem with electronic control units.

The reason for this phenomenon is twofold. Firstly, the lack of architecture support for the detection and identification of faulty components has a significant impact in the percentage of unidentified component failures. Secondly, the inability to duplicate the actual conditions that caused the failures in the laboratory is a cause of a high CND rate.

Avionics Domain

With the increasing use of electronic devices in avionics (e.g., fly-by-wire applications) the likelihood of a malfunction of an electronic component will also increase. Though reliability was subject to tremendous improvements during the last decades, transient failures reduce potential benefits. This is also recognized by James Pierce, former president of ARINC (quote taken from [Uni03]):

> *Despite significant improvements in mean time between failures of current avionics and the introduction of more comprehensive and accurate built-in test, the No Fault Found (NFF) ratio stubbornly hovers in the 50% area. This in effect halves the potential benefit of the technical advances made in avionics.*

According to [Uni03, Mah00] the NFF ratio is dramatically increasing. As stated, 50 to 60% of all in-flight avionics of commercial and military aircraft failures cannot be duplicated at ground. Since it is estimated that 30% of all avionics problems that are repaired are caused by component internal transient type defects, it can be concluded that the overall-rate of this type of failure is the number-one failure mode for electronic systems. This is reflected by the fact, that several depot LRU repair stations are reporting NFF rates over 90%.

These numbers are validated by a field study of Boeing, Texas Instruments and General Dynamics cited in [PR92], that reveals that 21 to 70% (depending on the type of system) of all failures could not be duplicated. Similar, the ROLM analysis

Failure Cause	Percentage
Parts	22%
No-defect-found	20%
Manufacturing defects	15%
Induced	12%
Wearout	9%
Design deficiencies	9%
Software	9%
System Management	4%

Table 3.2: Failure cause distribution of electronic systems in military aircraft

as cited in [Ram92] identified that 53% of boxes removed from the airplane are later found to be re-tested OK or cannot duplicate the fault. As a consequence, Boeing introduced an integrated systems approach for maintenance in its 777 airplane to reduce the no-fault found ratio [Ram92].

Dylis et al. [DP02] offer an interesting statistic from the Reliability Analysis Center (RAC) from the Department of Defense (DoD) regarding the failure cause distribution of electronic systems in military aircraft. The statistic reveals that 78% of failures originate from non-component causes with the distribution listed in Table 3.2.

Built-In Test Built-In Tests (BITs) are designed to identify operational conditions of electronic devices and facilitate automatic detection of wearout and breakage and has become an approved means of providing diagnostic support [GS02]. Despite the benefits of BITs there are still diagnostic deficiencies regarding the capabilities of identifying faults with minimal repeatability.

In [PDN+01] an example from the avionic domain is presented that underpins the fact that spurious fault detection is unacceptably high. The BIT of the Airbus A320 from Lufthansa had a daily average of 2000 error logs. Around 70 of these logs corresponded to faults reported by pilots, while another 70 pilot reports had no corresponding entry. Only two of the LRUs replaced every day, were found to have faults that correlated with the fault indicated by the BIT logs.

This example shows that fault detection is often incomplete and fault isolation inaccurate due to insufficient support of the underlying architecture. In combination with the ubiquitous tendency to treat testing of systems as a low priority task this results in a high rate of soft failures. The inability to reproduce the field environment responsible for the CND failures of the system is a determinant for the limited service possibilities of the ground service crews [PDN+01].

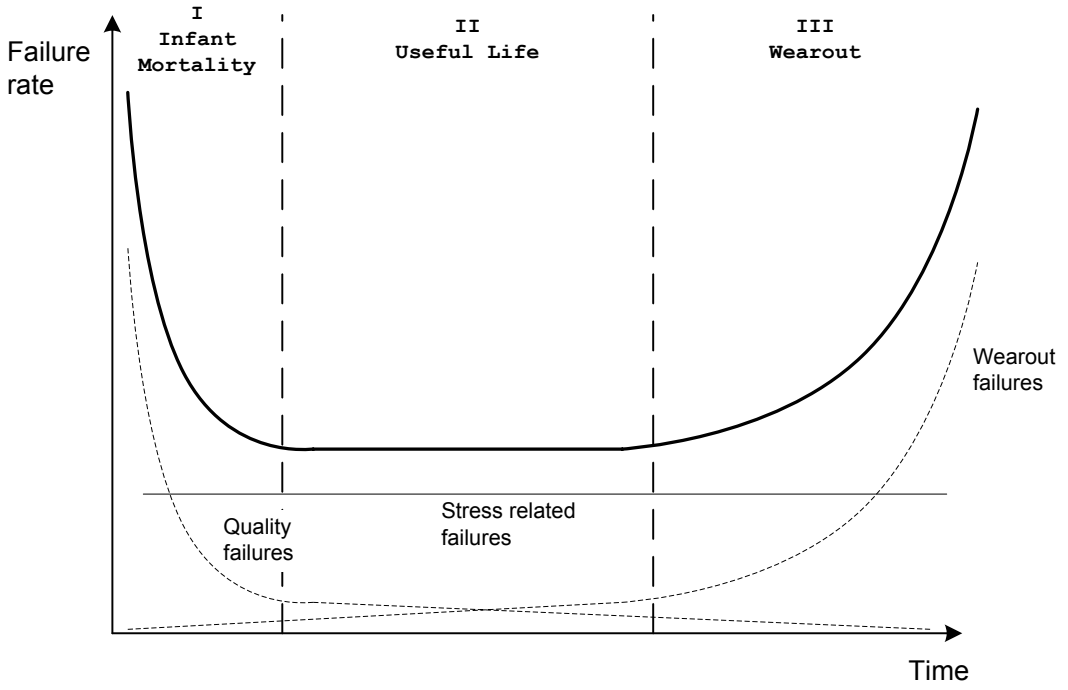

Figure 3.6: Bathtub Curve

3.5.2 Shift in Technology

Due to significant improvements of the IC industry during the last decades, the permanent failure rates of ICs is continually improving. The following section investigates this development and elaborates on the fact that the shift of technology also leads to a shift of the experienced failure modes of microelectronics. Various influence factors are considered and explanations for the latest developments are provided. At first, we describe the different phases of the predominant reliability model of electronic devices – the bathtub curve. In the second part the impact of emerging technology on the experienced failure modes is presented.

The Bathtub Curve

As depicted in Figure 3.6 the reliability of electronic components can be illustrated by the bathtub curve [DoD98, p. 5-28]. The bathtub curve is divided into three distinct phases, the infant mortality, the useful life, and the wearout phase. According to [Pec01] infant mortality failures are typically due to mistakes made during the manufacturing process. Thus, improved manufacturing can significantly reduce the incidence of such failures (i.e. fault avoidance).

Based on field data from the automotive industry Pauli and Meyna [PMH98, PM98] provide some very interesting facts on the failure rates during the period of infant mortality and useful life of the bathtub curve. In contrast to wearout

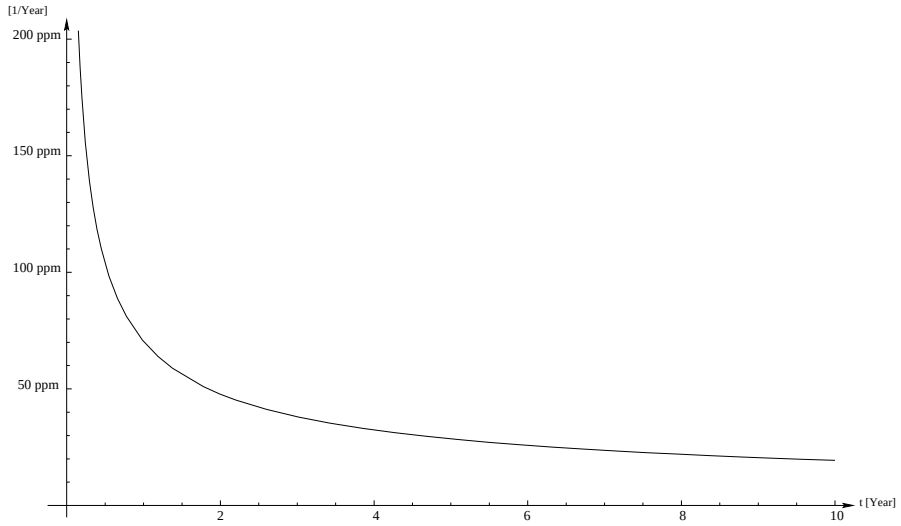

Figure 3.7: Time-dependent hazard rate $h(t)$ of an ECU (taken from [PMH98, p. 1014])

failures that affect the entire population, infant mortality failures tend to affect only a subpopulation of the shipped product [Pec01].

Figure 3.7 depicts the time-dependent failure rate of an ECU used in automotive applications. The reliability curve visualizes that the average failure rate of an ECU in the useful life period is very high. Reported failure frequencies are 50 out of 1 Million ECUs in 1 year. Harsh operating conditions with increased stress factors like temperature, shock and vibration, humidity, contaminants and radiation are affecting the reliability significantly [WWS99, IEE99]. The Pareto plot in Figure 3.8 shows the main reasons for ECU failure in the automotive domain.

Recent studies of ECUs used in automotive applications show that the parts of the components with the highest failure rates are PCBs and microcontrollers followed by analog ICs and ASICs (the higher the integration, the more likely the component is subject to fail). Resistors, transistors and diodes have the lowest failure rates (see also Figure 3.9). Capacitors are an exception since these components are more likely to be affected by aging processes and thus having a higher failure rate due to wearout than other discrete electronic elements. More detailed information on ECU failures an component failures can be found in [WWS99].

Failure mechanisms due to *accumulation of incremental damage beyond the endurance of the material* are termed wearout mechanisms [Ram01]. Early and premature wear-out failures are caused by the displacement of the mean and variability due to manufacturing, assembly, handling, and misapplication [Pec01]. Unfortunately, the wearout period is not covered by the study, since the manufacturers are only interested in field data during the warranty period of the product [PMH98]. Wearout due to the continuous use and stress of components is a natural phenomenon. Consider for instance the break pads of a car. According to the time of operation and operating conditions the abrasion of the pads is more or less advanced. The same is

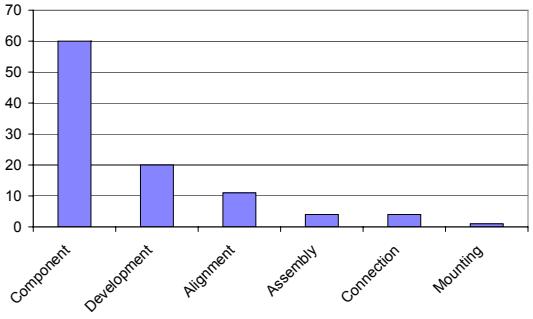

Figure 3.8: Pareto Plot of ECU Failures (taken from [PMH98, p. 1015])

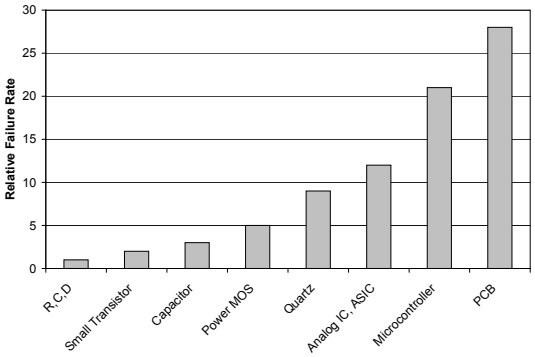

Figure 3.9: Components as Failure Causes in under-the-hood ECUs (taken from [WWS99, p. 2])

true for the profile of a tire (e.g., on the landing gear of an airplane). For many non electronic devices there exists the possibility of visual assessment of the condition of the equipment. Suitable indicators can be measured (e.g., depth of the remaining profile of a tire) and appropriate action can be taken by the service mechanics.

The question raises, whether we can find suitable indicators in the domain of electronic devices that allow to effectively and undoubtedly assess the condition of electronic devices. If advanced maintenance techniques like CBM are envisaged, then such indicators need to be identified.

A suitable indicator for wearout of electronic devices is the increase of transient failures in the system [Con02, BCGG97]. In fact, these spurious failures need to be analyzed to distinguish between random external transient disturbances (e.g., EMI) originating from outside the system and transient failures caused by internal faults (e.g., solder joint cracks, loose contacts).

A promising approach to monitor the condition of electronic circuits is presented in [MPG02, Goo00]. The purpose of so-called *prognostic cells* is to predict circuit failure. Such prognostic monitors are located on the same chip but are subjected to accelerated conditions to increase the rate of degradation relative to the companion functional circuit. This ensures that the monitor will fail before the main circuit. These prognostic monitors experience the same manufacturing process and the same environmental parameters. For this reason it its expected that the damage rate is the same for both circuits. Thus, this approach remedies the deficiencies of conventional off-line tests.

Impact of Emerging Technology on the Experienced Failure Modes

The tremendous improvements in the reliability of semiconductor devices is reflected in *Pecht's Law*. It suggests that semiconductor device reliability in terms of time-to-failure is doubling every fourteen months based on activation energy trends of semiconductor devices [MPG02].

This numbers are underpinned by Romanchik [Rom00] who states that in 1993 the failure rate for surface mount microprocessors in General Motor's engine control modules was nearly 450 parts per million (ppm). By 1997, that number hast dropped to less than 15 ppm. Based on these facts one can be misguided to believe that the reliability of modern microelectronics is in the magnitude needed to build dependable systems.

The types and causes of failures for electronics have changed over the years. Failure analysis in recent years has revealed that some failure causes may have been reduced by improvements in technology but due to the higher level of complexity and downsizing other failure classes have emerged [PR92].

According to Constantinescu [Con02] the primary cause for the significant increase of soft error rates are shrinking geometries, lower power voltages and higher frequencies. These result in higher sensitivity to neutron and alpha particles, and consequently have an impact on dependability by increasing the transient failure rates. Furthermore, due to semiconductor process variations and manufacturing residuals the likelihood of reoccurring permanent faults leading to transient failures is growing. The shrinking of geometries in semiconductor design has also significant impact on future design processes, such as nanometer design [LC02].

It can be summarized that the tremendous improvements made by the IC industry with respect to permanent failure are extenuated by increasing transient failure rates due to side effects of decreasing geometries of semiconductor technology. However, the statistics reveal little quantitative information on the transient failure rates of components.

3.5.3 Lasting Consequences on Business Realities

Electronic problems are not a phenomenon of low-cost or cheap cars. Quite on the contrary, luxury and high-end cars are primary affected by tricky electronic problems. Considering the complexity and amount of electronics in today's luxury cars this fact is no surprise.

Leadership in technical innovation is the prevalent claim of automotive manufactures in order to ensure customer loyalty and to differentiate the own products qualitatively from the competitors. However, the public discussion regarding the decreasing reliability of car's electronics in newspaper and non-scientific magazines has serious consequences of the customers opinion in the car manufacturer (e.g., [EW03]). The long term established cooperate image of luxury car manufacturers experiences lasting damage. This unwanted attention and accompanying image problems results in significant monetary expenses, either due to high warranty costs and/or the subsequent loss of customers.

However, it is important to note that all automotive manufacturers offering high-end technology are affected by this problem. In addition, the problem will delay the series production of emerging technologies, such as X-by-wire functionality. Ironically, these new technologies offer the possibility to introduce new electronic archi-

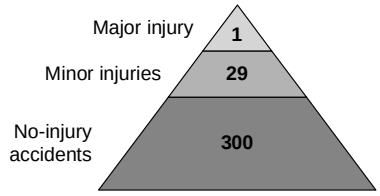

Figure 3.10: The Heinrich Pyramid

tectures into the car, that can significantly reduce the ratio of TNI problems by offering dependable error detection and isolation services (e.g., the Time-Triggered Architecture [KB03]).

3.6 The Heinrich Pyramid and Related Models

As a result of his studies on safety fundamentals, Heinrich proposed the so-called *Heinrich Pyramid* as the basis for his theory of accident causation [Hei50]. As depicted in Figure 3.10 for every major (fatal) injury there are 29 minor injuries and 300 no-injury accidents. This model proposes that the accumulation of accidents with low severity over time will cause incidents with higher severity in the future.

An adopted version of this model tailored to the avionic domain is presented in [Ran01]. The *air safety information model* as illustrated in Figure 3.11 identifies four types of possible occurrences: significant accidents, incidents, aircraft defects, and unreported occurrences. It is important to note, that the more serious the incident has been, the more information on this incident is available. From an economic point of view it is understandable, that a thorough investigation of incidents implying no consequences for the passengers or the aircraft at all, is not performed. This is in contrast to accidents causing (fatal) injuries or severe damage to the aircraft.

The main objective of the investigation of aircraft accidents and incidents is the determination of the root cause in order to prevent future similar occurrences. It is important, to identify whether a human error or a failure of a technical subsystem caused the accident or incident. In case the latter applies, aircrafts of the same type are also affected by the imminent danger of experiencing the same problems.

It can be concluded that in the avionic domain the peak of the pyramid is covered very well, whereas the amount of available information of the other layers of the model are decreasingly covered.

With the increasing use of electronic systems in the avionics and automotive domain it is important to focus on the identification of anomalies in the fundament of the pyramid. By applying the air safety information model pyramid to electronic systems it can be assumed that the increasing rate of system anomalies is a suitable indicator to forthcoming component failure. The increasing number of TNI phenomena as elaborated on in Section 3.5 emphasize the need for diagnostic infrastructures that take system anomalies into account in order to shrink the upper layers of the

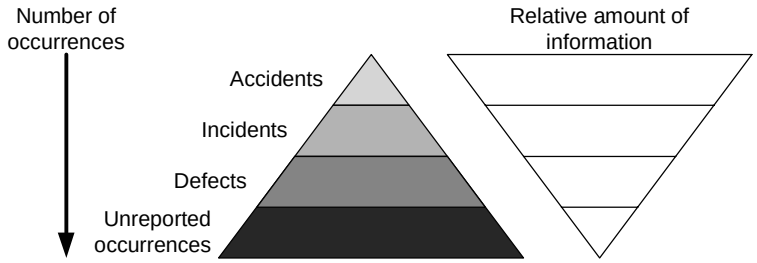

Figure 3.11: Air Safety Information Model based on the Heinrich Pyramid

pyramid (excluding human errors). Therefore, a diagnostic infrastructure must take these system anomalies into account.

3.7 Analysis Techniques

Numerous analysis techniques for diagnosis can be found in the literature. In the following we will briefly discuss prevalent analysis techniques, in particular threshold-based algorithms, probabilistic networks, and model-based diagnosis methods.

3.7.1 Threshold-Based Techniques

In the following we describe the basic principles and rationale behind threshold-based analysis algorithms and describe architectures relying on this analysis techniques.

Alpha-Counter

The rationale for the α-count mechanism is to decide on the point in time when keeping a system component on-line is no longer beneficial [BCGG97]. The algorithm is partly based on the observation that intermittent (transient internal) faults exhibit a relatively high occurrence rate after their first appearance. The α-count is a threshold-based fault classification mechanism designed to identify permanent faulty components from components affected by external transient faults. The main idea of the algorithm is to keep track of every fault occurrence in each component. When the α-counter value exceeds a given threshold value, the component is diagnosed as affected by a permanent/intermittent fault. Depending on the expected frequency of permanent, intermittent and transient faults the values assigned to the parameters of the algorithm are set.

In [BCGG97] the basic algorithm is defined as follows. Let $J_i^{(L)}$ indicate the L-th judgment on a component u_i: then $J_i^{(L)} = 0$ means that the error detection mechanisms deployed at component u_i has detected no violation of the specification, while $J_i^{(L)} = 1$ denotes that the error detection mechanisms has detected a failure. The α-count algorithm associates a score α_i to each component u_i to record information

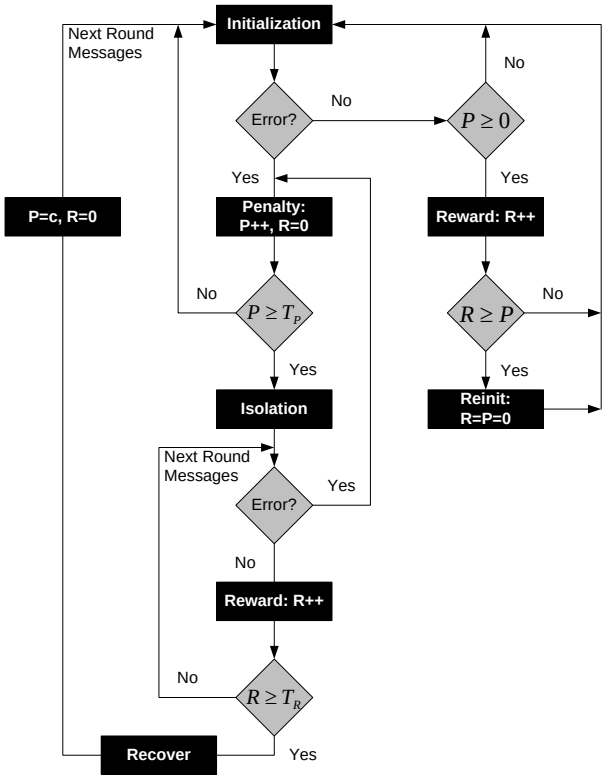

Figure 3.12: Two Level Threshold-Based Algorithm

about experienced failures affecting the component and allow judgment about the health status. α_i is initialized to 0 and accounts for the pertinent L-th judgment the following way:

$$a_i^L = \begin{cases} a_i^{(L-1)} \cdot K & \text{if} \quad J_i^{(L)} = 0 \\ a_i^{(L-1)} + 1 & \text{if} \quad J_i^{(L)} = 1 \end{cases} \qquad 0 \leq K \leq 1$$

In case a_i^L exceeds an a priori defined threshold value α_T, the component u_i is judged as being permanent faulty. The outcome of the α-count mechanism heavily depends on the parameters K and α_T.

In [GCB98] an improved version of the α-count mechanism with two instead of only one threshold value is presented resulting in better performance of the algorithm. Such a two level threshold scheme is depicted in Figure 3.12. Two thresholds, T_P and T_R are used. While the first one specifies when a node is considered to be faulty (no longer classified as transient faulty, but permanent), the threshold value T_R defines when a faulty node is declared healthy again. This way, nodes analyzed as faulty during the first analysis process can be classified as healthy again, provided that no more errors of the component are detected. Consequently, the most important design parameters of the algorithm are the penalty threshold T_P and the reward threshold T_R. Furthermore, a reintegration penalty value, denoted c in Figure 3.12, can be

parameterized. A comprehensive overview on the different design choices for the parameter settings can be found in [BCGG00]. The exact values of these parameters is application specific and depends on the application field, field data, and experience of the system designer.

Architectures using Threshold-Based Analysis Techniques

Multicomputer Architecture for Fault-Tolerance The Multicomputer Architecture for Fault-Tolerance (MAFT) is designed to comply to the performance requirements in the avionics domain (e.g., flight control system) [KWFT88].

A MAFT system consists of several node computers connected by a broadcast bus network. Each node is partitioned into two separate processors, namely the operations controller and the applications processor. The operations controller is designed to handle the vast majority of the system's executive functions, for instance, communication and synchronization, data voting, error detection, task scheduling, and system reconfiguration.

Diagnosis in MAFT. The operations controller of a node continuously monitors the messages of all other nodes of the MAFT system. Once an error is detected by the error detection mechanisms of the operations controller, a corresponding error message is generated (up to 31 error flags). In addition, a penalty counting mechanism is used to assess the condition of each node of the system. Whenever an error is detected, a penalty weight, unique to the triggered detection mechanisms, is added to the error counter. The weight of the penalty is determined by the application designer and is intended to reflect the criticality of the error. This penalty count mechanism is also used to communicate the overall health of the node.

By exchanging this information periodically, Byzantine agreement on the error counters is guaranteed. Afterwards the counters are updated and compared to a predefined exclusion threshold. In case the value exceeds the given threshold, the operations controllers will propose to exclude the faulty node from the system. Similarly, by exchanging this information, Byzantine agreement on the operational state of each node is achieved.

In the MAFT architecture, excluded nodes remain functional but will not be included into the computational activities of the other nodes. Therefore, in case the node was excluded due to a transient fault, the node will not accrue penalties as soon as the nodes exhibits correct behavior. At predefined points in time (master period), the counter will be decremented for a specific value (or zero in case for permanent exclusion from the system). When the nodes fall below a given threshold, the operations controller decide via the same sequence on inclusion.

A faulty detection mechanism is identified by the fact that the generated error report differs from the consensus. In order to accelerate the detections of faults within the detection mechanism, self-test mechanisms are used.

Generic Upgradable Architecture for Real-Time Dependable Systems Generic Upgradable Architecture for Real-Time Dependable Systems (GUARDS) [PABD+99] is designed for ultra-dependable real-time systems. The design rationale sets emphasis on design for validation, reuse of pre-validated components and support of software modules with different criticalities.

Fault Classes. GUARDS is designed to tolerate permanent and transient physical faults and should provide tolerance of software design faults. The architecture distinguishes between temporary external physical faults (also called transients) and temporary internal faults (also called intermittents). The latter are treated as either permanent or transient faults according to their rate of recurrence.

Mechanisms. The collection of error reports generated during the interactive consistency and consolidation exchanges forms the first step in the diagnosis process in GUARDS. Subsequently, the messages are filtered to assess the criticality of the detected errors and to determine further action if necessary.

The filtering of the information is done by applying the previously introduced α-count mechanism [BCGG97]. In GUARDS a distributed version of the α-count assessment method is used to allow diagnosis of channels, i.e. a shared multi-processor system built from commercial off-the-shelf components. Once a channel is diagnosed to be faulty and consequently excluded from the system, the channel is isolated and reset. A self-test is used to determine whether a transient or permanent fault is affecting the channel. In case of a permanent fault the channel is shut down and subject to maintenance.

3.7.2 Probabilistic Networks

Probabilistic networks [CDLS99, Cha91] are gaining increasing popularity for diagnostic applications. Especially, in traditional artificial intelligence domains like medicine the use of probabilistic or *Bayesian networks* has a long history as the data structure of choice for expert systems. See for instance [KJ03] for a recent medical application of Bayesian networks for an analysis of infertility data. Due to its universal nature the use of probabilistic networks is not limited to particular application domains. For instance, in [SJK00] a Bayesian network approach is implemented to help in troubleshooting printer problems. Similarly, in [PDT03] a system is described that allows the diagnosis of field replaceable units on the basis of Bayesian networks and inference. A survey about algorithms for Bayesian inference can be found in [GH02].

Bayes' Theorem

Bayes's Theorem is defined as

$$P(T|E) = \frac{P(E|T) \times P(T)}{P(E)}$$

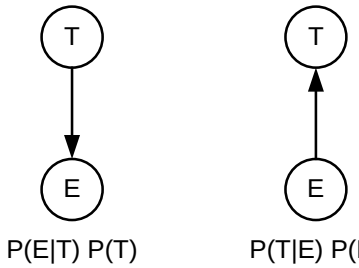

Figure 3.13: Bayesian Inference [CDLS99]

where T stands for theory and E for evidence. Since E is equivalent to $(E \wedge T) \vee (E \wedge \neg T)$ and the two disjuncts are mutually exclusive, we can write $P(E) = P(E \wedge T) + P(E \wedge \neg T)$. This way we get the following version of the Bayes' Theorem:

$$P(T \mid E) = \frac{P(E \mid T) \times P(T)}{P(E \mid T) \times P(T) + P(E \mid \neg T) \times P(\neg T)}$$

The generalized version of the theorem, where the probability of a particular theory T_k out of a collection of alternatives $T_1, T_2, \ldots, T_n$ is

$$P(T_k \mid E) = \frac{P(E \mid T_k) \times P(T_k)}{\sum_{i=0}^{n} P(E \mid T_i) \times P(T_i)}$$

and is typically applied when evaluating a theory in relation to a number of competitors (Note, that the competitors must be mutually exclusive).

As depicted in Figure 3.13 Bayesian inference can be graphically represented by a directed arrow indicating the *cause* and *effect* relationship [CDLS99].

Bayesian Networks

As stated in [CDLS99] the main idea of a graphical model is to allow experts to concentrate on building up the qualitative structure of a problem before beginning to address issues of quantitative specification. In graphical probability models nodes represent random variables, and arcs represent conditional independence assumptions [Mur01]. One can distinguish two fundamental graphical models, namely undirected (also known as Markov networks) and directed ones (Bayesian networks). In directed graphical models, an edge from one node X_i to another node X_k can be informally interpreted as indicating that X_i *causes* X_k.

Formally, a Bayesian network for a set of random variables $\mathbf{X} = X_1, \ldots, X_n$ consist of [Hec95]:

1. a network structure S that encodes a set of conditional independence assertions about variables in $\mathbf{X}$

2. a set P of local probability distributions associated with each variable

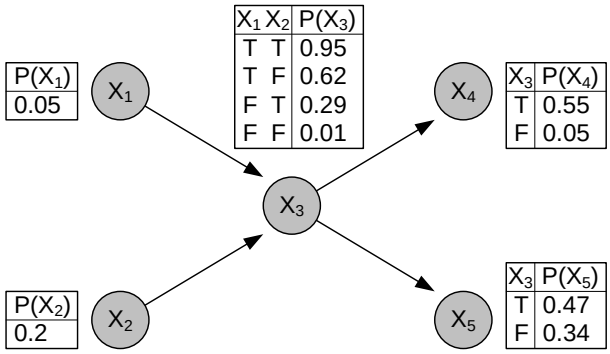

Figure 3.14: Bayesian Network Example

Together, both S and P define the joint probability distribution for $\mathbf{X}$. The network structure S is a directed acyclic graph, where the nodes in S are in one-to-one correspondence with the variables $\mathbf{X}$. Given the network structure S, the joint probability distribution for $\mathbf{X}$ is given by

$$P(\mathbf{X}) = \prod_{i=1}^{n} P(X_i \mid \mathbf{Pa}_i)$$

where each X_i denotes both the variable and its corresponding node and $\mathbf{Pa}_i$ denotes the parents of node X_i in S and the variables corresponding to those parents.

According to [DvdG00] the design of a probabilistic network involves three tasks: the identification of variables of importance, the definition of the relationship between theses variables expressed in a graphical structure, and finally, to obtain the probabilities that are required for its quantitative part. This probabilistic information typically comes from statistical data (e.g., field data), literature, or human experts in the application domain [DvdG00].

An exemplary Bayesian network is depicted in Figure 3.14. This network consist of five nodes ($\mathbf{X} = X_1, X_2, X_3, X_4, X_5$). X_1 and X_2 are both causing X_3, which itself shows the symptoms X_4 and X_5 with the probabilities given in the following (e.g., derived from field data):

$$P(X_1) = 0.05 \quad P(X_2) = 0.2$$

$$\begin{aligned} P(X_3 \mid X_1 \wedge X_2) &= 0.95 \\ P(X_3 \mid \neg X_1 \wedge X_2) &= 0.62 \\ P(X_3 \mid X_1 \wedge \neg X_2) &= 0.29 \\ P(X_3 \mid \neg X_1 \wedge \neg X_2) &= 0.01 \end{aligned}$$

$$\begin{aligned} P(X_4 \mid X_3) = 0.55 \quad P(X_4 \mid \neg X_3) = 0.05 \\ P(X_5 \mid X_3) = 0.47 \quad P(X_5 \mid \neg X_3) = 0.34 \end{aligned}$$

In a Bayesian network if one wants to know the probability of a node X_i, only the parent nodes (i.e. the causes) are of interest. For the calculation, all possible

combinations of the values of the parent nodes need to be stated. For example, if one needs to know the probability of $P(X_3)$ then we need to calculate the probability of each case that can lead to node X_3 and add the probabilities:

$$\begin{aligned} P(X_1 \wedge X_2 \wedge X_3) &= P(X_3 \mid X_1 \wedge X_2) \times P(X_1) \times P(X_2) \\ P(\neg X_1 \wedge X_2 \wedge X_3) &= P(X_3 \mid \neg X_1 \wedge X_2) \times P(\neg X_1) \times P(X_2) \\ P(X_1 \wedge \neg X_2 \wedge X_3) &= P(X_3 \mid X_1 \wedge \neg X_2) \times P(X_1) \times P(\neg X_2) \\ P(\neg X_1 \wedge \neg X_2 \wedge X_3) &= P(X_3 \mid \neg X_1 \wedge \neg X_2) \times P(\neg X_1) \times P(\neg X_2) \end{aligned}$$

Then $P(X_3)$ is just the sum:

$$P(X_3) = 0.95 \times 0.05 \times 0.2 + 0.62 \times 0.95 \times 0.2 + 0.29 \times 0.05 \times 0.8 + 0.01 \times 0.95 \times 0.8$$

3.7.3 Model-Based Diagnosis

In order to determine that the behavior of a component deviates from the specified one, either a priori knowledge or redundancy is necessary. In case of model-based techniques, the use of *analytical redundancy* is the main idea to detect and diagnose component failures [CW84]. Measurements from different senors are compared analytically on the basis of a mathematical model describing their relationship. The differences are called residuals. In an ideal scenario the residuals are zero. However, in practice both noise and faults result in deviations from zero. By the use of statistical analysis, it is possible to decide wich residuals can be considered as normal and which residuals are due to a fault. Finally, by analyzing the patterns derived from the residuals a failure cause analysis is made possible. Figure 3.15 depicts these three stages of model-based diagnosis, residual generation, statistical testing, and logical analysis [Ger88].

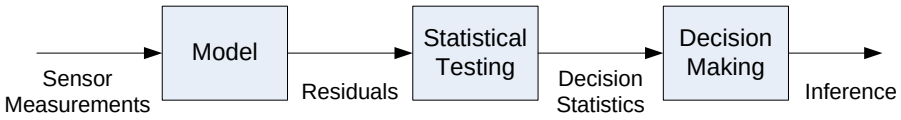

Figure 3.15: Model-Based Diagnosis Stages

The advantage of the model-based approach is the fact that the used models can be directly devised from the design of a new device. Furthermore the models are considered to be task independent, i.e. the same model can be used for diagnosis and other tasks such as simulation. For this reason, model-based diagnosis is gaining a lot of industrial interest. For example, in [SC97] the model-based diagnosis approach of the Fiat car company for automotive repair is described.

In [GCF+95] a model-based diagnostic algorithm for an automotive engine system is presented. The work is motivated by environmental regulations that require on-board detection of malfunctions that may affect the vehicle's emission performance. For this reason the introduced diagnostic algorithm aims to detect and diagnose faults affecting two actuators (i.e. the fuel injectors and exhaust gas recirculation valve)

and the four sensors of the engine (i.e. the throttle position, the manifold pressure, the engine speed, and the exhaust oxygen). The model-based method continually compares the measured values of the system input/outputs to the respective mathematically modeled values to determine wether the system is functioning properly. For this purpose parity equations, derived from the input-output equations of the monitored system, are employed. Each of the parity equations returns a residual that becomes nonzero in case of faults and disturbances. In order to tolerate noise and modeling errors, the residuals are tested against nonzero threshold values. For fault detection a single residual is typically sufficient, however, for determining the source a set of residuals are required. Other examples for model-based diagnosis in the context of automotive engines are presented in [Nyb02] and [CWGW04]. In [Nyb02] the author discusses a diagnosis solution for the air-intake system of an engine, while in [CWGW04] a model-based monitoring solution for an electronic throttle control system is introduced.

For an excellent introduction and more detailed information on model-based detection and analysis techniques see [Ger88] and [PW03].

Chapter 4

System Model: The DECOS Integrated Architecture

Depending on the physical structuring of large distributed safety-critical real-time systems, one can distinguish federated and integrated system architectures. In a federated system, each application subsystem has its own dedicated computer system, while an integrated system is characterized by the integration of multiple application subsystems within a single distributed computer system. Federated systems have been preferred for ultra-dependable applications due to the natural separation of application subsystems, which facilitates fault-isolation and complexity management. Integrated systems, on the other hand, promise massive cost savings through the reduction of resource duplication. In addition, integrated systems permit an optimal interplay of application subsystems, reliability improvements with respect to wiring and connectors, and overcome limitations for spare components and redundancy management.

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

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

- the strong competition among the carmakers;

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

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

In the following section we present the DECOS (Dependable Embedded Components and System) integrated architecture for dependable embedded control systems, as developed in the European Sixth Framework Programme [KOPS04]. This integrated architecture is based on a time-triggered core architecture and a set of high-level services that support the execution of newly developed and legacy applications across standardized technology-invariant interfaces. Rigorous encapsulation guarantees the independent development, seamless integration, and operation without unintended mutual interference of the different application subsystems. The integrated architecture offers an environment to combine both safety-critical and non safety-critical subsystems within a single distributed computer system. The architecture exploits the encapsulation services to guarantee that software faults cannot propagate from non safety-critical subsystems into subsystems of higher criticality.

4.1 Physical and Functional Structuring

Until now, the introduction of *structure and hierarchical relationships* represents the only promising approach for understanding complex systems with large numbers of parts and interactions between these parts [Sim96, chap. 8]. This insight applies to all technical systems and in particular to the mastering of large, complex real-time computer systems. The complexity of a large real-time computer system can only be managed, if the overall system can be decomposed into nearly-independent subsystems with linking interfaces that are precisely specified in the value and time domain [KS03a]. Near-independence is the ability of a subsystem to serve its purpose independently from the detailed structure of other subsystems, i.e. only based on the specification of the linking interfaces of the subsystems.

As a consequence, in the DECOS architecture, one can distinguish between physical and functional structuring of the integrated system. While physical structuring is concerned with hardware entities such as components and wiring, the functional structuring provides rules and guidelines for grouping the distributed applications in such a way that an optimal interplay of the constituting functional entities is guaranteed.

4.1.1 Functional System Structuring

For the provision of application services at the controlled object interface, the real-time computer system is divided into a set of nearly-independent subsystems, each providing a part of the computer system's overall functionality. We denote such a subsystem as a *Distributed Application Subsystem (DAS)*, since the implementation of the corresponding functionality will most likely involve multiple components that are interconnected by an underlying communication system. The implementation as a distributed system is a prerequisite for establishing fault-tolerance by redundantly performing computations at separate components that fail independently. Furthermore, a distributed solution becomes a necessity, when the resource requirements of the application providing the subsystem's functionality exceed the available resources of a single component.

In analogy to the structuring of the overall system, we further decompose each DAS into smaller units called *jobs*. A *job* is the basic unit of work that employs the communication system for exchanging information with other jobs, thus working towards a collective goal. The interface between a job and the communication system is denoted as a *port*. Depending on the data direction, one can distinguish input ports and output ports. A job employs input ports for exploiting the services of other jobs, while output ports enable a job to provide its own services. Every job has access to its relevant transducers, either directly via the controlled object interface or via a communication system with known temporal properties.

Automotive Examples

In order to cope with complexity today's automotive systems are typically split up into several domains. The powertrain domain includes all necessary functionality for engine, transmission, and active safety (e.g., ESP, ABS) management. The comfort/body domain covers the interior of a car (e.g., seats, doors), the passive safety domain the airbag control, and the infotainment domain the telematics and in-car entertainment systems. By using the concept of DASs this domain structuring can be reduced to smaller subsystems that have a solely a functional coherence, and not also a physical one (e.g., sharing the same CAN bus for economic reasons). In the following we describe two examples that form DASs.

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

  In by-wire applications, a deterministic behavior of all safety-related message transmissions must be guaranteed even at peak-load. Time-triggered communication protocols can provide this deterministic behavior. In addition to hard

real-time performance they support temporal composability and dependability. As a consequence, time-triggered architectures are widely accepted as the computing infrastructure for future by-wire cars [HT98].

- **Brake-by-wire DAS.** Brake-by-wire [Bre01] is a DAS that controls the braking of the car. In addition, it improves the braking functionality of conventional anti-lock braking systems. Brake-by-wire systems remedy deficiencies of conventional hydraulic braking systems, such as aging of braking fluids, difficulties routing of pipes, the inconvenient feedback during ABS braking. Brake-by-wire systems incorporate brake power assist, vehicle stability enhancement control, parking brake control, and tunable pedal feel.

Avionics Examples

To show that the concept of a DAS can be used in any type of system, we describe in the following some examples of DASs in avionics.

- **Cabin Pressure System.** The purpose of the cabin pressurization subsystem is to control the cabin pressure to the required value depending on the aircraft altitude by regulating the flow of air from the cabin. In long-range passenger aircraft, the system regulates the cabin pressure, so that a cabin altitude of about 8000 feet is never exceeded. In case of an unintended decompression, the control system must ensure, that oxygen masks deploy to provide the passengers with oxygen while the pilot initiates an emergency descent [MS03].

- **Primary Flight Control.** The primary flight control system controls the yaw, pitch, and roll rate of an aircraft. Typically, the roll control is invoked by using the right and left ailerons, yaw control by means of two or three rudder sections, and pitch control by powering four elevator sections [MS02]. Fly-by-wire aircraft deploy an ultra-dependable computer systems instead of a conventional hydraulic/mechanic system as the primary flight control system [Col99]. However, all modern civil aircraft using fly-by-wire systems employ in addition some form of direct mechanical link as a back-up system. The primary-flight control system must provide a dependability of 10^{-9} failures per hour [SWH95].

- **In-flight Entertainment.** As the typical passenger profile changed over the years, a sophisticated in-flight entertainment system deployed on an aircraft is an important aspect for a successful business. Entertainment systems intended for avionics use are fundamentally different from the equipment used for home entertainment. For example, the entertainment equipment needs to withstand aircraft temperature changes, air pressure changes, and vibration. In addition, electromagnetic susceptibility and power quality cannot be compared to those deployed for home use [Lee98]. State-of-the art in-flight entertainment systems comprises digital media servers and several hundred interconnected

seat electronic boxes [LK00]. Despite the wiring complexity, high-end entertainment systems, such as currently deployed on the 777 aircraft, incorporate about 250.000 lines of code [Bax97].

- **Flight Data Recording.** The flight data recording subsystem is mandatory on every commercial aircraft of a certain size. Such a system has demanding requirements with respect to shock, fire, and long-term immersion in seawater. A flight data recording subsystem comprises its own sensors and wiring to perform its intended functionality. Modern aircraft have a flight data recording subsystem that is capable of recording more than 80 parameters [MS03].

4.1.2 Physical System Structuring

During the development of an integrated system the functional elements must be mapped to the physical building blocks of the platform. These building blocks are *clusters*, *physical networks*, *components* and *partitions*. A *cluster* is a distributed computer system that consists of a set of components interconnected by a *physical network*. A *component* is a self-contained computational element with its own hardware (processor, memory, communication interface, and interface to the controlled object) and software (application programs, operating system) [KS03a], which interacts with its environment by exchanging messages across LIFs. The behavior of a component can be specified in the domains of value and time. Components are the target of job allocation and provide encapsulated execution environments denoted as *partitions* for jobs. Each partition prevents temporal interference (e.g., stealing processor time) and spatial interference [Rus99] (e.g., overwriting data structures) between jobs. In the DECOS architecture, a component can host multiple partitions and host jobs that can belong to different DASs.

4.1.3 Namespace of the Integrated Architecture

Following this structuring of the DECOS architecture, the namespace of the integrated architecture consists of a part reflecting the physical structure and a part reflecting the functional structure of the system:

$$\underbrace{id_{\text{cluster}}.id_{\text{component}}}_{\text{physical structure}} \;:\; \underbrace{id_{\text{subsystem}}.id_{\text{DAS}}.id_{\text{job}}}_{\text{functional structure}}$$

The physical part identifies the cluster (id_{cluster}) and the component ($id_{\text{component}}$) of the integrated architecture, while the functional part following the colon identifies the subsystem ($id_{\text{subsystem}}$), the DAS (id_{DAS}), and the job (id_{job}), where id is the numerical identification of a structuring element. The identification is statically assigned by the system integrator at design time. Table 4.1 exemplifies the introduced notation. For convenience, it is possible to omit either the physical or the functional part if not needed by the specification. For example, by describing only functional aspects of the safety-critical subsystem, one can abstract from the physical allocation

Notation	Meaning
:0.3.4	job 4 of DAS 3 of subsystem 0 (abstracting from physical structure)
0.2:0.3.4	job 4 of DAS 3 of subsystem 0, in component 2 of cluster 0
:0.1.*	all jobs of DAS 1 of subsystem 0 (abstracting from physical structure)
:0.1	DAS 1 of subsystem 0 (abstracting from physical structure)
:0.*.*	all jobs of all DAS of subsystem 0 (abstracting from physical structure)
0.3:	component 3 of cluster 0 (abstracting from functional structure)
0.*:	all components of cluster 0 (abstracting from functional structure)

Table 4.1: Examples for the introduced notation of the namespace

(i.e. writing only $:id_{\text{subsystem}}.id_{\text{DAS}}.id_{\text{job}}$). However, within the part wildcards (*) are used to express that a particular *id* is not of interest, but all elements are referred to. For example, $:id_{\text{subsystem}}.id_{\text{DAS}}.*$ addresses all jobs of a particular DAS of a particular subsystem.

Based on this namespace, we also introduce a source-based namespace for message. $m(id_{\text{cluster}}.id_{\text{component}}:id_{\text{subsystem}}.id_{\text{DAS}}.id_{\text{job}})$ identifies the message that is produced by job id_{job} of DAS $:id_{\text{subsystem}}.id_{\text{DAS}}$ in component $:id_{\text{cluster}}.id_{\text{component}}$.

4.1.4 Architectural Services

Generic architectural services separate the application functionality from the underlying platform technology in order to facilitate reuse and reduce design complexity. This strategy corresponds to the concept of platform-based design [SV02], which proposes the introduction of abstraction layers, which facilitate refinements into subsequent abstraction layers in the design flow.

The DECOS architectural services depicted in Figure 4.1 are such an abstraction layer. The specification of the architectural services hides the details of the underlying platform, while providing all information required for ensuring the functional and meta-functional (dependability, timeliness) requirements in the design of a safety-critical real-time application. The architectural services serve as a validated stable baseline that reduces application development efforts and facilitates reuse, because applications build on an architectural service interface that can be established on top of numerous platform technologies.

In order to maximize the number of platforms and applications that can be covered, the DECOS architectural service interface distinguishes a minimal set of *core services* and an open-ended number of *high-level services* that build on top of the core services. The core services include predictable time-triggered message transport, fault tolerant clock synchronization, strong fault isolation, and consistent diagnosis of failing components through a membership service. The small number of core services eases a thorough validation (e.g., permitting a formal verification), which is crucial for preventing common mode failures as all high-level services and consequently all applications build on the core services. Any architecture that provides

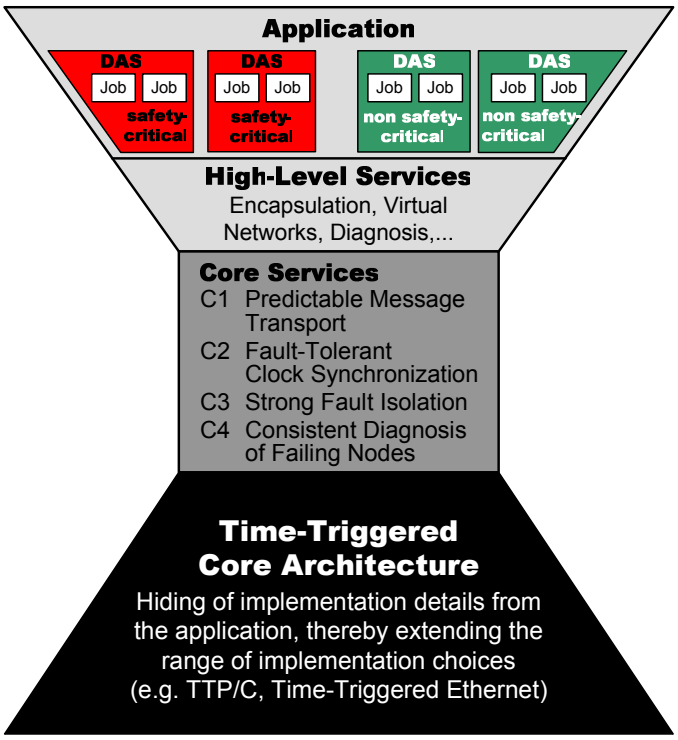

Figure 4.1: The DECOS Integrated System Architecture

these core services can be used as a core architecture [Rus01b] for the DECOS integrated distributed architecture. An example of a suitable core architecture is the Time-Triggered Architecture (TTA) [KB03].

Based on the core services, the DECOS integrated architecture realizes high-level architectural services, which are DAS-specific and constitute the interface for the jobs to the underlying platform. Among the high-level services are virtual network services and encapsulation services. On top of the time-triggered physical network, different kinds of virtual networks can be established and each type of virtual network can exhibit multiple instantiations (see Figure 4.1). The encapsulation services ensure spatial and temporal partitioning for virtual networks in order to obtain error containment and control the visibility of exchanged messages.

4.2 Core Services

The core services are centric to the architecture, since high-level services build on top of these core services. These core services include a time-triggered transport service, clock synchronization, fault isolation, and consistent component diagnosis.

Deterministic and Timely Transport of Messages

The purpose of this service is the transport of state messages from the Communication Network Interface (CNI) of the sending component to the CNIs of the receiving components. The fault-tolerant transport service is offered by a time-triggered communication service that is available via the temporal firewall interface [KN97], which eliminates control error propagation by design and minimizes coupling between components. The timely transport of messages with minimal latency and jitter is crucial for the achievement of control stability in real-time applications. While control algorithms can be designed to compensate a known delay, delay jitter (i.e. the difference between the maximum and minimum value of delay) brings an additional uncertainty into a control loop that has an adverse effect on the quality of control [KK90].

Fault-Tolerant Clock Synchronization

Due to clock drifts, the clock times in an ensemble of clocks will drift apart, if clocks are not periodically resynchronized. Clock synchronization is concerned with bringing the values of clocks in close relation with respect to each other. The clock synchronization is a fundamental service in a time-triggered system, since all activities are controlled by the progression of time [KO87].

Strong Fault Isolation

Although a Fault Containment Region (FCR) can demarcate the immediate impact of a fault, fault effects manifested as erroneous data can propagate across FCR boundaries. For this reason, the system must also provide error containment [LH94] as introduced in Section 2.5.3. To avoid error propagation by the flow of erroneous messages the error detection mechanisms must be part of different FCRs than the message sender. Otherwise, the error detection service can be affected by the same fault that caused the message failure. The set of FCRs that perform error containment is denoted as an ECR [Kop03]. In the DECOS architecture a component is considered to be a FCR with respect to hardware faults. An ECR must consist of at least two independent FCRs. In the DECOS architecture, an ECR is constructed via a component and one of the replicated central guardians.

Consistent Diagnosis of Failing Nodes

The component-level membership service provides consistent information about the operational state (correct or faulty) of nodes [Cri91]. The membership service is based on the a priori knowledge about the points in time of the time-triggered message exchanges. In a time-triggered system the periodic message send times are the membership points of the sender [Kop97]. Every receiver knows a priori when a message of a sender is supposed to arrive, and interprets the arrival of the message as a life sign at the membership point of the sender. From the arrival of the expected

messages at two consecutive membership points, it can be concluded that a node was operational during the interval delimited by these membership points [BP00].

4.3 High Level Services

Based on the core services, high-level services are constructed that can be exploited by all deployed applications running on a DECOS system. The high-level services are the encapsulation service, the virtual networks service, the gateway service, the fault-tolerance service, and the diagnosis and maintenance service.

4.3.1 Encapsulation Service

The encapsulation service is responsible for spatial and temporal error containment [Rus99] at the component-level. The component-level error containment occurs in two phases:

- **Error Containment between the Safety-Critical and the Non Safety-Critical Subsystem.** As depicted in Figure 4.1, the overall system comprises a safety-critical and a non safety-critical subsystem. The encapsulation service statically allocates component resources (i.e. memory and processor) to these two subsystems. Such a statically defined allocation facilitates certification [RTC99]. Furthermore, control and information flow may occur only from the safety-critical subsystem to the non safety-critical subsystem, but not the other way around.
- **Error Containment between Jobs at each Subsystem.** The encapsulation service manages the access of each job to component resources. The high-level encapsulation service ensures that each *job* executes within a protected partition within the respective subsystem and restricts interactions between jobs to the ports. Thus, the high-level encapsulation service ensures error containment between jobs. The encapsulation service is also the key mechanism for Intellectual Property (IP) protection.

4.3.2 Virtual Network Service

A virtual network is an overlay network that is established on top of a physical network. In the DECOS integrated system architecture, we provide virtual networks on top of the time-triggered core communication service of the base architecture. To achieve the advantages of the federated approach in an integrated architecture, we propose the provision of a dedicated virtual network for each DAS in order to exchange messages between the jobs of the DAS. Each virtual network is tailored to the requirements of the respective DAS via the provided functionality, the operational properties, the namespace, and the dependability properties. Furthermore,

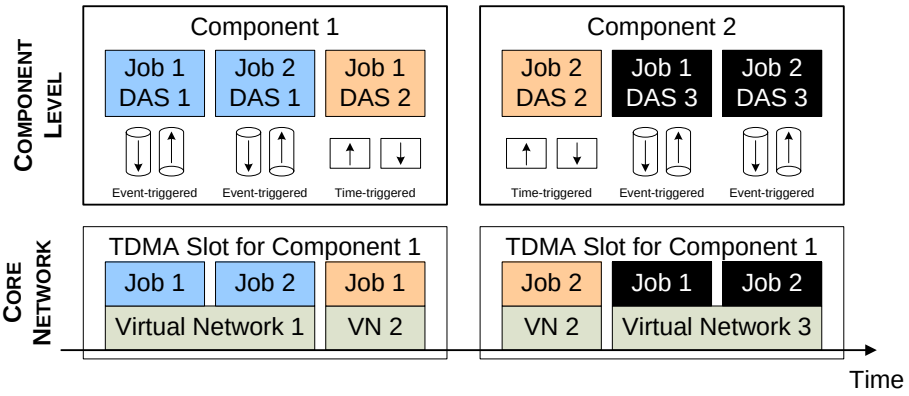

Figure 4.2: Virtual Network Service

each virtual network is encapsulated so that communication activities in other virtual networks are neither visible nor have any effect (e.g., performance penalty) on the exchange of messages in the virtual network. Consequently, virtual networks extrapolate the idea of component-level error containment to the network level [OPK05b].

Figure 4.2 depicts the realization of virtual networks in the DECOS architecture. The available bandwidth of each time-triggered frame disseminated via the core network is split up according to the hosted DASs and respective jobs of a component. This way, the timely exchange of messages is guaranteed and the exclusive use of the bandwidth precludes any interference between virtual networks. In case of time-triggered virtual networks a state message interface is used, while for even-triggered virtual networks a queueing interface is deployed.

Time-Triggered Virtual Networks

Time-triggered virtual networks interconnect the jobs of the safety-critical DASs. The virtual networks for the safety-critical DASs are strictly time-triggered, because of the respective advantages of the time-triggered control paradigm (e.g., with respect to predictability, error detection, fault-tolerance, replica determinism [Kop95, Rus01a]). Of course, time-triggered virtual networks can also be employed for the non safety-critical subsystem relying on state information A time-triggered virtual network is designed for the periodic exchange of state messages. The self-contained nature and idempotence of state messages eases the establishment of state synchronization, which does not depend on exactly-once processing guarantees.

Event-Triggered Virtual Networks

In analogy to time-triggered virtual networks, event-triggered virtual networks are the communication infrastructure for DASs with jobs communicating via event-triggered ports. An event-triggered virtual network enables each job of a DAS to disseminate event messages that are being received by the other jobs in the DAS.

An event-triggered virtual network is designed for the sporadic exchange of event messages, combining event semantics with external control. While in time-triggered virtual networks a state interface is used, in event-triggered virtual networks queues are used as the interface to the network.

Interface Specification

The operational specification of a DAS in the integrated DECOS architecture occurs at three levels:

- **Port Specification.** A port is dedicated to the transmission or reception of message instances of a single message. Each port has a statically defined data direction, thus forming either an input port or an output port. The port specification captures the syntactic and temporal properties of the message instances of the received or the sent message. Only those temporal properties are part of the port specification, which are defined for the port in isolation,i.e. independently from other ports (i.e. local constraints).

- **Link Specification.** The link of a job consists of the ports provided to the job. The link specification contains the respective port specifications and additional temporal properties that can be defined only with respect to multiple ports of the job (i.e. global constraints). An example for the additional temporal properties would be a statement for the latency between the reception of a request message at an input port and the transmission of the corresponding reply message at an output port of the job.

- **Virtual Network Specification.** The virtual network specification consists of all link specifications in the DAS and those temporal properties that can be defined only with respect to ports of more than one job.

4.3.3 Gateway Service

Even with a strict separation of DASs, the integrated DECOS architecture would already permit a considerable reduction in the numbers of components and wiring through the sharing of components and networks among DASs. However, controlled interactions between DASs are required for unleashing the full advantages of the integrated approach.

On the one hand, the quality of control of a real-time computer system can be improved when different control functions are coordinated to achieve a tactic behavior. In the automotive industry, an example of the coordination of different application subsystems for improving the quality of service with respect to passenger safety is the passive safety mechanism (Pre-Safe) of the Mercedes S-class [Bir03]. The Pre-Safe system tensions seat-belts, realigns seats to a safer position, and closes an open sun

roof when sensors detect possibly hazardous situations. The system correlates information of existing car dynamics sensors in order to determine hazardous situations such as skidding, emergency braking, or avoidance maneuvers.

Secondly, in a large real-time computer system, different DASs will typically depend on the same or similar sensory inputs and computations. By adapting encapsulation to allow for the exporting and importing of information between DASs, one DAS can use services (e.g., acquisition of sensory information or computations) in the other DASs and does not need to provide the services on its own. For example, in an automotive system the speed sensors from the factory installed Antilock Braking System (ABS) can be exploited to estimate the car's heading for the navigation system during periods of GPS unavailability [CG02]. The redundant sensors can be eliminated in one of the DASs leading to reduced resource consumption and hardware cost. Alternatively, redundancy can be exploited to improve the reliability of the sensory information. Even sensory information from different physical entities can be exploited by sensor fusion [Elm02].

Consequently, a gateway in the DECOS architecture offers two key funcationalities [OPK05a], namely *property transformation* and *encapsulation*. In general, the semantic and operational properties of the input ports at one virtual network can be different to the semantic and operational properties of the output ports at the other virtual network. The resulting *property mismatch*, i.e. a disagreement among connected interfaces in one or more of their properties, is resolved by the gateway by performing transformations on the information passing through the gateway.

Secondly, in general, only a fraction of the information exchanged at one virtual network will be required by jobs interconnected via the gateway. By restricting the redirection through the gateway to the information actually required by the jobs of the other DAS controlled export and import of information can be established. This way, the gateway not only improves resource efficiency by saving bandwidth of unnecessary messages, but also facilitates complexity control.

Central to the construction of a gateway is the *gateway repository*, i.e. a real-time database storing the message contents from the two virtual networks to be interconnected via the gateway. The gateway repository needs to ensure temporal accuracy [Kop97] of the stored state messages and manages respective queues in case of event messages. For a more detailed analysis about gateways in the DECOS architecture refer to [OPK05a].

In addition to the interconnection of virtual networks, the presented integrated architecture offers architectural gateway services for interacting with the environment via *physical gateways*. In general, the interaction of the computer system with the controlled object and the human operator can occur either via a direct connection to sensors and actuators or via a fieldbus network. The latter approach simplifies the installation – both from a logical and a physical point of view – at the expense of increased latency of sensory information and actuator control values. Since the prevalent low-cost fieldbus protocol in the automotive domain is LIN [Fle03], the proposed architecture supports physical LIN gateways, each acting as a master for

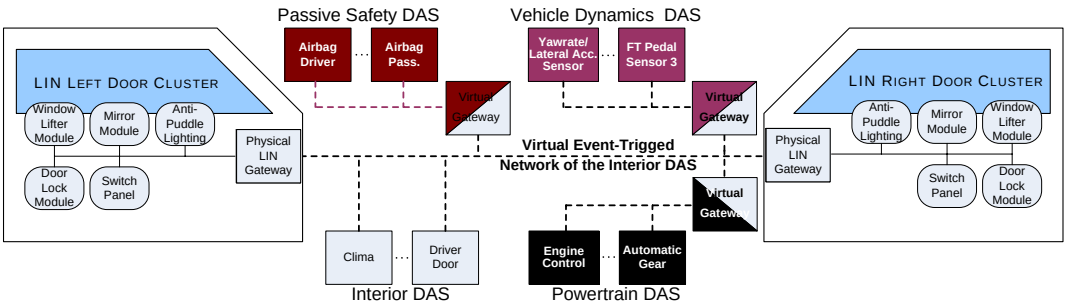

Figure 4.3: Physical and Virtual Gateways in an Automotive Application

the slaves of the physical LIN bus. Figure 4.3, which exemplifies the role of hidden gateways in the functional structure of the future automotive architecture, depicts two of these LIN gateways for the interconnection of the interior DAS with the LIN fieldbusses located at the front doors [POT+05]. The driver door job and passenger door job exchange information with the actuators and sensors in the doors in order to control door locks, window lifters, mirrors, and anti-puddle lighting.

In addition, Figure 4.3 also contains virtual gateways for the interconnection of virtual networks. The interior DAS constructs real-time images capturing the state of the passenger compartment of the vehicle (e.g., status of the doors, seats, lighting, climate control) that are also important to other DASs. For example, the driver's weight as measured at a seat is an important parameter for the passive safety DAS in order adapt air bags to different passengers (e.g., children). The current temperature inside and outside the car, which is captured by the climate control subsystem, is another example for a real-time entity that is significant beyond the interior DAS. Temperature measurements are an essential input for physical models of sensors in other DASs (e.g., powertrain DAS) and permit to improve the precision and plausibility of sensory information. Adversely, other DASs need to be able to control body electronics in the interior DAS. In hazardous situations, e.g., after the detection of a potential crash as indicated by yaw rate and lateral acceleration sensors (e.g., during skidding and emergency braking), the vehicle dynamics DAS causes the tensioning of seat-belts, and realigning of seats to a safer positions.

4.3.4 Fault-Tolerance Service

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

A common approach for masking component failures is N-modular redundancy (NMR) [RLT78, Avi75, LA90, Sch90]. N replicas receive the same requests and provide the same service. The output of all replicas is provided to a voting mechanism, which selects one of the results (e.g., based on majority) or transforms the results to

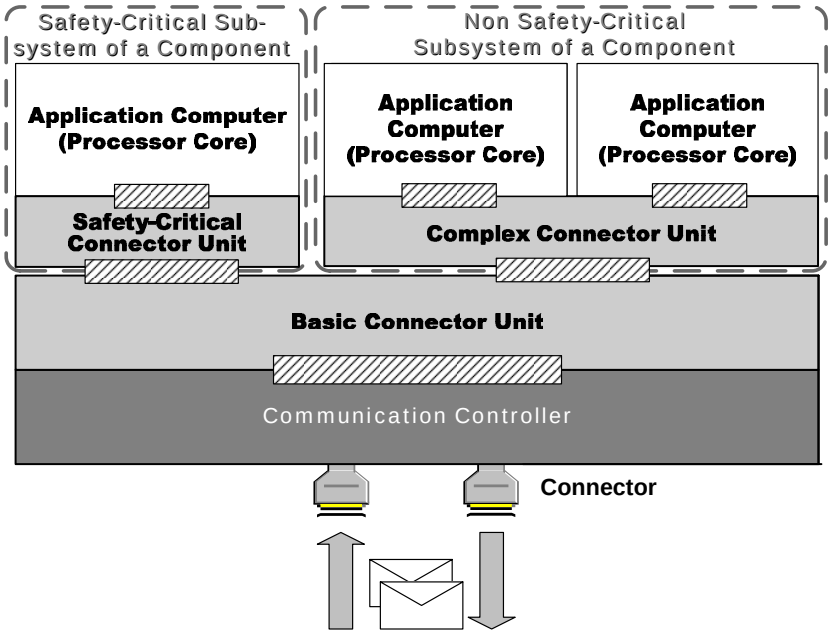

Figure 4.4: The DECOS Component Model

a single one (average voter). The most frequently used N-modular configuration is triple-modular redundancy (TMR). By employing three replicated jobs and a voter, a single consistent value failure in one of the constituting jobs can be tolerated. Note that the assignments of jobs to components must ensure that jobs fail independently.

The major strength of group masking is the ability to handle component failures systematically at the architecture level, i.e. transparently to the application. A requirement for this systematic fault-tolerance is the support for replica determinism [Pol95b], both by the architecture and the application. The purpose of the fault-tolerance services of the architecture is the provision of mechanisms required for managing the redundant groups of replicas in a way that masks component failures and makes the group functionally indistinguishable from a single replica.

4.3.5 Diagnosis and Maintenance Service

The high-level diagnostic service is responsible for establishing diagnostic information exceeding the component-level granularity of the core diagnostic service and will be discussed in detail in the following Chapter 5.

4.4 Component Structure

In the DECOS component model as depicted in Figure 4.4 we distinguish between two kinds of structuring, *horizontal* and *vertical* structuring. The vertical structuring

of the component provides two subsystems within a component. The *safety-critical subsystem* is an encapsulated execution environment for ultra-dependable applications. The *non safety-critical subsystem* offers an environment for those applications having less stringent dependability requirements. For these applications, emphasis lies on low-cost, flexibility, and resource efficiency. The safety-critical and non safety-critical subsystems are established by means of *spatial* and *temporal* inner-component partitioning [Rus99].

The software in the communication controller and connector units is triggered solely by the progression of real-time on the sparse time base [Kop92]. This triggering mechanism is the key element for ensuring determinism of the architectural services, which is a prerequisite for the determinism of the overall system and thus for fault-tolerance through active redundancy and exact voting. In addition, the consistent distributed state induced by the sparse time base simplifies state recovery and diagnosis.

The vertical and horizontal structuring of a component is primarily driven by certification concerns. Since support for modular certification [Rus01c] is a major requirement for integrated architectures, the strict separation of the architectural services from applications as well as the separation of different DAS and jobs is also maintained at the component level. This separation of the certification efforts allows for the construction of independent safety arguments [BB98] for different DASs.

As depicted in Figure 4.4 the interconnection between the communication controller and the application computers occurs via *connector units*, which control the application computers access to the state message interface provided by the communication controller. The primary purpose of a connector unit is the allocation of network resources within a component that is vertically structured into two or more subsystems. The connector unit ensures that each subsystem obtains a predefined share of the overall network resources, thus enabling each subsystem to exchange messages with guaranteed temporal properties (maximum latency and latency jitter of message transmissions, minimum bandwidth) and data integrity.

The DECOS architecture does not restrict the choice of implementations of a particular component. Although the provision of separate processors (or processor cores) for each partition has significant advantages with respect to certification, a solution with only one processor shared among the jobs is also an alternative choice as long as the operating system providing the partitioning can be certified up to the highest criticality class [RTC92].

In the DECOS component model we distinguish between three types of connector units (see Figure 4.4). The *Basic Connector Unit (BCU)* performs the primary allocation of the physical network resources, as required for the separation of the safety-critical and non safety-critical subsystems of a component.

The *Saftey-Critical Connector Unit (SCU)* allocates network resources to the jobs of the safety-critical subsystem and realizes the safety-critical high-level services (e.g., voting functionality). In analogy to the BCU, simplicity of the safety-critical connector unit is of major concern in order to control certification efforts.

The *Complex Connector Unit (XCU)* performs the allocation of network resources for the non safety-critical subsystem of a component. Like the SCU, the XCU does not directly access the communication controller, but builds on top of the BCU. This way, the XCU is not involved in the fault isolation and error containment between the safety-critical and non safety-critical subsystem of a component, as this separation is performed by the underlying BCU. Therefore, the XCU and the non safety-critical subsystems of a component need not be certified to the highest criticality levels and the XCU can provide increased functionality at the cost of increased complexity. In contrast to the SCU, the XCU can support the time-triggered and the event-triggered control paradigm. The event-triggered paradigm facilitates the establishment of cost-effective solutions for the non safety-critical subsystems. In this domain, the high flexibility and resource efficiency of event message ports outweigh the lower predictability and increased complexity resulting from the event-triggered control paradigm with its imprecise temporal specifications (e.g., probabilistic queuing models). Furthermore, legacy applications following the event-triggered paradigm can be reused without involving redevelopment efforts. This legacy integration helps in leveraging investments in existing software and lays ground for evolving systems.

For a more detailed description (e.g., constituting hardware elements and software modules) and an analysis of this model with respect to certifyability, encapsulation and independent development aspects refer to [KOPS04].

4.5 Design Flow

The design flow of automotive distributed systems can be decomposed into three phases, the requirement analysis, the subsystem design, and the system integration phase [GFL+02] (see also Figure 4.5). As described in [RH04, SW04] an ECU-centric design process prevails in the automotive industry. Such a bottom up process, however, bears significant drawbacks such as resource duplications, local instead of global quality-of-service optimization, and exponential growth in terms of system integration costs. Furthermore, the number of the deployed ECUs steadily increases to satisfy recent market trends and the customer's demand for new functionality.

The DECOS architecture, by contrast, also supports a top-down design approach. During the requirement analysis the system integrator captures the requirements of the overall system (i.e. the car electronics) and decomposes the system into nearly-independent subsystems (i.e. DASs). The requirement analysis provides the foundation for all later design stages. Here, the overall functionality of the system is specified and subsystems are identified to enable an independent development of DASs. As depicted in Figure 4.5 the result of this design phase is a set of DASs that comprise the electronic infrastructure of the car.

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

1. **Functional Coherence.** A DAS should provide a meaningful application service (e.g., brake-by-wire service of a car) to its users at the controlled object

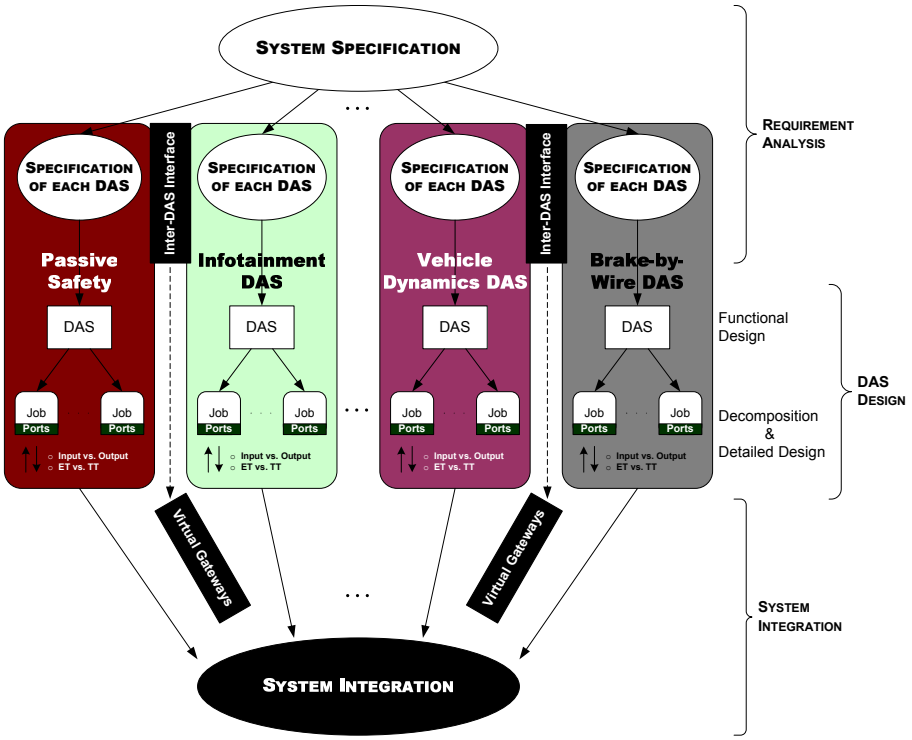

Figure 4.5: Design Flow

interface. By associating with a DAS an application service that is relevant in the actual application context, the mental effort in understanding the various application services is reduced. An application service can be analyzed by solely considering the jobs of the DAS, the interactions to the controlled object and the gateways to other DASs (inter-DAS interfaces). In particular, it is not necessary to possess knowledge about the internal behavior of DASs, other than the one providing the application service that is of interest.

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

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

triggered or event-triggered) and a corresponding configuration (e.g., bandwidth).

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

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

In order to ensure correct system integration the specification of inter-DAS relationships is of high importance. Inter-DAS interfaces as indicated in Figure 4.5 are used to specify common information within DASs (e.g., sensor information), possible interrelationships via the controlled object, and meta-functional aspects. This way, resources can be shared among DASs, thus avoiding resource duplication by eliminating sensors or using redundant sensory information to improve dependability.

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

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

4.6 Dependability

This section describes the fault hypothesis of the integrated system architecture. The fault hypothesis states the assumptions about the units of failure, the failure modes and the failure frequencies that a fault-tolerant system must tolerate. The units of failure are denoted as Fault Containment Regions (FCRs) [Kop03]. A FCR is considered as a subsystem that shares one or more common resources that can be jointly affected by a single fault. Consequently, a FCR is the delimiter of the immediate impact of a fault. In the integrated system architecture, we perform a

differentiation of FCRs for hardware and software faults. The distinction of hardware and software faults is part of the fault discrimination introduced in [ALR01].

4.6.1 Hardware Fault Model

A hardware fault hits physical resources, such as mechanical or electronic parts. Hardware faults originate either from development or from conditions that occur during operation. Hardware design faults and production defects belong to the first class. The second class includes physical deterioration (i.e. aging) and external interference through physical phenomena (e.g., lightning stroke).

For hardware faults, we regard a component, i.e. a complete node computer constituting a hardware/software unit, as a FCR. Since a component contains shared physical resources (e.g., processor, memory, power supply, oscillator), a single physical fault hitting any of these resources is likely to jointly affect several or all of the jobs within the component. Furthermore, we assume that hardware diversity mechanism are applied to prevent common mode failures [KBJ00] due to developmental hardware faults (e.g., replicated production defects, hardware errata). This approach is widely accepted for safety-critical applications, e.g., in the avionic domain [Yeh98, BT93].

The failure mode of a hardware FCR is assumed to be arbitrary. The failure frequency in case of permanent hardware failures is in the order of 100 FIT [PMH98]. In case of transient failures a significantly higher failure frequency in the order of 1000-10000 hours is assumed.

The fault hypothesis assumes that only a single hardware FCR becomes faulty within a bounded interval of time. Further, a new component may become faulty only after the previously faulty one (if any) either has shut down or operates correctly again.

4.6.2 Software Fault Model

For software faults, we regard a job as a FCR. If a job is replicated along multiple components as part of a fault-tolerance concept, the FCR includes all physically distributed replicas of the job. Replicated jobs cannot be assumed to fail independently, since all replicas of a job are based on the same programs and use the same input data. An example of an FCR consisting of replicated instances of a job is depicted in Figure 4.6.

The failure mode of a job is a violation of the port specification in either the time or value domain. In case of a failure in the value domain, the content of a message does not conform to its specification, while in case of a timing failure, the send instant of the message is incorrect.

The role of jobs as software FCRs holds also in case of software diversity. When software diversity is applied for addressing common mode failures, replicas are necessarily different and ideally employ different specifications in addition to separate im-

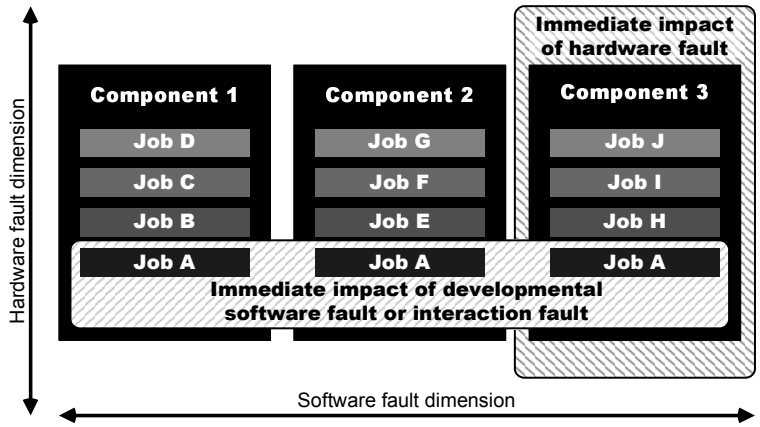

Figure 4.6: Immediate Impact of a Developmental Software or Interaction Fault (Horizontally Expanding Section-Lined Box) and Immediate Impact of a Hardware Fault (Vertically Expanding Section-Lined Box)

plementations. Consequently, we denote these diverse replicas as separate jobs. Nevertheless, the decision of regarding these jobs as different software FCRs depends on the independence of the diverse software version. Practical analyses [ALS88, LPS01] of software diversity have demonstrated that diverse implementation exhibit correlation.

The identification of FCRs for the proposed integrated system architecture demonstrates the differences in the boundaries of the immediate impact of different types of faults. As depicted in Figure 4.6, the expansion of FCRs for hardware and software faults proceeds along two dimensions. In case of replicated jobs, software faults hit multiple components and affect well-delimited subsystems of these components, namely the partitions housing the replicated instances of the job. The FCR for hardware faults, on the other hand, expand within a component and we assume that hardware faults are delimited by a component. This assumption is justified in case of hardware diversity and if precautions are taken to avoid common mode failures (e.g., due to spatial proximity, common ground).

4.6.3 Distinction between Heisenbugs and Transient Hardware Faults

Gray [Gra86] divided software faults into Bohrbugs and Heisenbugs. Bohrbugs are design errors in the software that cause reproducible failures (e.g., a logic error in a program). In contrast to Bohrbugs, which can be identified during testing, Heisenbugs are design errors in the software that seem to generate quasi-random failures. An example for a Heisenbug is a synchronization error that causes the occasional violation of an integrity condition.

From a phenomenological point of view, a failure that is caused by a Heisenbug cannot be distinguished from a failure caused by a transient hardware malfunction.

Experience shows that it is much more difficult to find and eliminate Heisenbugs than it is to eliminate Bohrbugs from a large software system.

TMR as the primary fault-tolerance mechanism of the integrated architecture (see Section 4.3) is designed to handle both transient hardware faults and Heisenbugs. The latter under the assumption that the replicas will not exhibit correlated Heisenbugs at the same time.

From a diagnostic point of view a discrimination of failures with respect to the originating fault is significant, i.e. whether the failure was caused by a Heisenbug or a transient hardware fault. The replacement of a component with a component of the same type will be ineffective for removing a Heisenbug. Here, only an update of the deployed software will eliminate the spurious malfunctions permanently. For this reason, diagnostic mechanisms operating on the distributed state are needed in order to reveal correlated failures. In addition, statistical field data of a large population (e.g., car model) will provide valuable input that allows to determine whether software shows correlated failures as a consequence of undiscovered software faults, i.e. Heisenbugs.

Chapter 5

Diagnosis Model: An Integrated Diagnostic Architecture

In accordance with the differentiation of FCRs for hardware and software faults introduced in Section 4.6, we introduce a fault model that identifies a component as the FRU with respect to hardware faults and a job as the FRU for software faults under the assumption that the architectural services are free of design faults. In addition, the integrated diagnostic services of the architecture support the maintenance engineer in the identification of faulty connectors (replacement of FRU vs. replacement of connectors). In order to cope with industry demands on diagnosis, the

- *detection* and subsequent
- *identification* of the FRU(s) causing malfunction

must be supported by the system architecture. In particular, the system needs to support the maintenance engineer by providing a health status indication for each FRU that acts as a foundation for the decision process whether a FRU remains in the system or will be replaced. This online analysis of the gathered diagnostic information is mandatory for future generations of computer systems to reduce the numbers of cannot duplicate failures, i.e. failures that cannot be reproduced at the service station.

5.1 Requirements for an Integrated Solution

An integrated diagnostic architecture, i.e. the provision of generic architectural services as part of the architecture, has many advantages in comparison with isolated diagnostic subsystems at every component. The access to information that is typically hidden from the applications can be exploited for lowering the probability of wrong recommended maintenance actions. In order to tackle diagnostic problems that industry is currently facing, an integrated solution must satisfy the following requirements:

- **Service Technician Assistance.** As already stated in Section 3.3 diagnostic deficiencies of deployed architectures complicate the job of the service technician to remove only those components that are causing the system malfunction. Since a mechanic at a service station is no specialist in electronics, the diagnostic system must provide all necessary information that allows maintenance of faulty components. If this is not possible, fully operational units will be replaced by mistake. This is especially important when X-by-wire solutions will be subject to mass production, since computer diagnostic will become a standard part of the job [LH02, Bre01].

- **Focus on Transients.** The types and causes of failures for electronics have changed over the years (see Section 3.5). Failure analysis in recent years has revealed that some failure causes may have been reduced by improvements in technology but due to the higher level of complexity and downsizing other failure classes have emerged [PR92].

  The increase of transient failures in combination with software design faults due to the growth of software complexity of modern electronic systems leads to substantial difficulties in diagnosing electronic problems (commonly referred to as *TNI*, *CND* or *NFF* failures) [TAP02, MRS+02, Bar01].

- **Detection of Correlated Errors.** Diagnostic systems operating on only the local internal component states preclude the possibility to detect and analyze correlated failures or system anomalies. An integrated diagnostic architecture must thus provide means to establish a holistic view on the system by operating on the distributed state. This includes also context information such as environmental parameters (e.g., temperature) that can give important hints in the identification of spurious anomalies. For example, a software bug in the electronic management unit of the fuel pump of a luxury car caused the car to stall in case the fuel tank was below 1/3 full. By correlating the information of the fuel pump unit in case of an engine failure with the distributed state of the related ECUs, the fault can be traced or at least valuable information that helps to trace the malfunction can be provided to the maintenance engineer. Consequently, a pivotal property of any diagnostic infrastructure must be the recording of the state of the system at the time of occurrence of a behavior deviating from the expected service. Such an on-the-fly analysis is vital, since many faults are not active at the service station, which renders comprehension through the service technician impossible. Furthermore, the operation on the distributed state has the significant advantage of allowing an independent assessment of the services of independent FCRs.

- **Reduction of Complexity.** Typically more than 50% of the code deployed in an automotive ECU is specific to diagnosis [PBC+02]. By realizing a substantial fraction of this overhead by the use of generic architectural services, significant economic benefits during system design can be expected. Furthermore, since diagnosis plays an integral role during the design flow, developers

are forced to deal with maintenance issues from the beginning on, resulting in a more structured developmental process instead of ad-hoc solutions.

- **Assessment of Fault-Tolerance Mechanisms.** Fault-tolerance mechanisms are required to achieve the necessary degree of dependability for the deployment of electronic systems in safety-critical environments. In order to reduce application complexity and certification efforts, fault-tolerance mechanisms are in the best case provided by the architecture and exploited transparently to the application [Bau01]. However, from a diagnostic point of view, this strategy has far reaching implications. It is impossible to detect inconsistency of the fault-tolerant replicas at application level. Consequently, architectural diagnostic services need to monitor the health state of the fault-tolerance mechanisms. Furthermore, the diagnostic subsystem must provide means to support the inspection of fault-tolerance mechanisms – also known as scrubbing techniques – to validate the functionality of fault-tolerance mechanisms in specified time intervals (cf. [Aer93]).

- **Support for Condition-Based Maintenance.** TBM is increasingly being replaced by CBM, to reduce costs and to improve reliability and system performance [TS01]. As already introduced to the avionics domain, this new paradigm is becoming more and more accepted in the automotive industry. Besides the reduction of cost of ownership (service only what is needed) the possibility of collecting accurate field data (i.e. engineering feedback) is one of the major benefits from this maintenance approach. In addition, the customer trust in the car will be increased, since component replacements can be performed by the service technicians before the owner of the car is informed by the car's OBD system. In order to adopt CBM for electronic systems suitable indicators for degradation or wearout must be identified and analyzed to detect deviations from sound operation.

- **Certification Support.** In case diagnosis is part of the application every change of the diagnostic code will cause a recertification of the application software in case of ultra-dependable applications. When handling diagnostics at the architectural level, a recertification of the application is rendered obsolete in case of changes of the diagnostic subsystem.

- **Avoidance of the Probe Effect.** Any diagnostic subsystem must avoid the introduction of probe effects [Gai86] that may forge the outcome of the diagnostic subsystem. This is especially important in case of real-time systems, where the diagnostic messages must not compromise the real-time traffic in any way. In addition, the diagnostic subsystem is required to sustain the reliability level of the system. Any additional hardware (e.g., wiring, connectors) may be subject to failures that would not be existent without the diagnostic subsystem.

- **Intellectual Property Protection.** Diagnosis is often equated with revealing of internal information. An integrated approach allows the realization of

advanced diagnostic strategies without revealing any internals of the application by solely operating on the interface state of the linking interfaces. Consequently, applications of different vendors can be joined without changing the source code. All that is needed are precise interface specifications that allows the discrimination of the behavior with the intended service.

- **Identification of Rare Failures.** During product development and testing low quality issues are relatively easy to identify because they are uncovered with smaller sample size. The problem of current normal vehicle testing process is the identification of statistically very unlikely occurring incidents, that become only identifiable after high volume production [GW02]. Since an extensive vehicle level test with larger test fleets is impractical given current economic and time constraints, continuous online diagnosis can reduce the probability of undetected (design) flaws.

- **Legal Issues.** [Bre01] addresses an emerging problem introduced by new technology in the automotive industry. With the use of X-by-wire applications in automotive systems legal issues in case of an accident need to be clarified. The question "who is to blame" in a lawsuit can only be answered, if a recording device like the flight data recorder (black box) in the avionic's domain is used. The recording of all communication activities within the safety-critical system forms the basis to determine whether a human failure, a technical failure or a combination of both caused the accident. These investigations are also necessary for claiming compensation from the insurance company.

5.2 Overview and Strategy

An architecture with integrated support for diagnosis provides the necessary prerequisites to allow the effective detection, identification and classification of experienced errors. The model of the diagnostic architecture as illustrated in Figure 5.1 can be divided into two main parts, the acquisition of diagnostic information via a dedicated *virtual diagnostic network* and the subsequent analysis located in a dedicated *diagnostic DAS* in order to determine the nature of an experienced fault with respect to a *maintenance-oriented fault model*.

The pivotal strategy of the diagnostic architecture is the establishment of a holistic view on the system by operating on the *distributed state*. In combination with the inter-component and inner-component error containment mechanisms provided by the basic and high-level services, this strategy allows to trace correlated system anomalies back to the FRU responsible for the experienced system behavior. As depicted in Figure 5.2 one can classify diagnostic solutions according to the deployed detection and analysis mechanisms. While prevalent component local solutions have less stringent requirements with respect to the underlying platform, global solutions allow a more accurate determination of the required maintenance action. Furthermore, diagnostic mechanisms at architecture level have access to all relevant interface

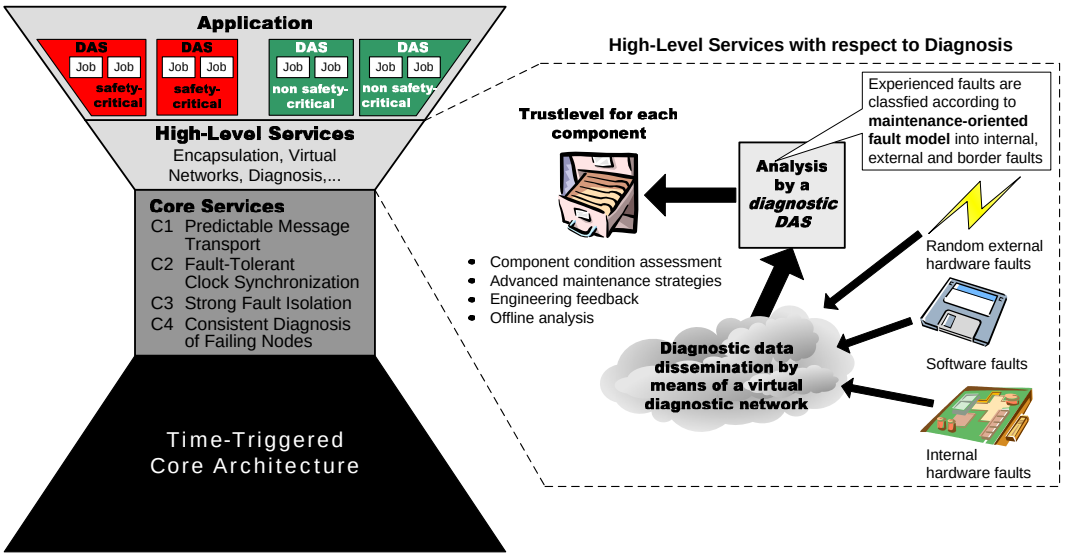

Figure 5.1: Overview of the Diagnostic Infrastructure

state variables of the distributed system, while at application level certain state variables are inaccessible and thus precluded from analysis. In contrast to the internal component states, the distributed state can be *independently checked* by the diagnostic architectural services. Such a detection is thus much more trustworthy than any internal check that cannot be independently verified. Consequently, this strategy provides the foundation for solving prevalent diagnostic problems, in contrast to diagnostic systems operating only on the local internal state (e.g., like most OBD system currently deployed in the automotive industry).

The diagnostic framework decouples *architecture level* diagnosis from *application level* diagnosis to reduce the efforts of the application developers. Generic mechanisms parameterized by the application developers eliminate the need for the deployment of proprietary solutions. The systematic diagnosis techniques (e.g., identification of component hardware failures) are independent of a particular application and need not be covered by the respective application diagnosis strategy (e.g., plausibility checks). In addition, a revalidation of the systemic diagnosis mechanisms by the manufacturer is rendered obsolete if the coverage of the deployed mechanisms has been validated and can be reproduced deterministically.

From a service technician's point of view, the diagnostic solution can ultimately be reduced to the question whether a FRU should be replaced or remain in the system. In the integrated diagnostic architecture the analysis subsystem, i.e. the diagnostic DAS, provides a *trust level* for each component, that acts as the foundation for the decision of the maintenance engineer on the replacement.

In the following we will discuss the constituting elements of the diagnostic architecture of the DECOS integrated system. In Section 5.3 the maintenance-oriented fault model that forms the conceptual foundation is introduced. All deployed mechanisms of the diagnostic architecture are based on this model. An extensive suitability

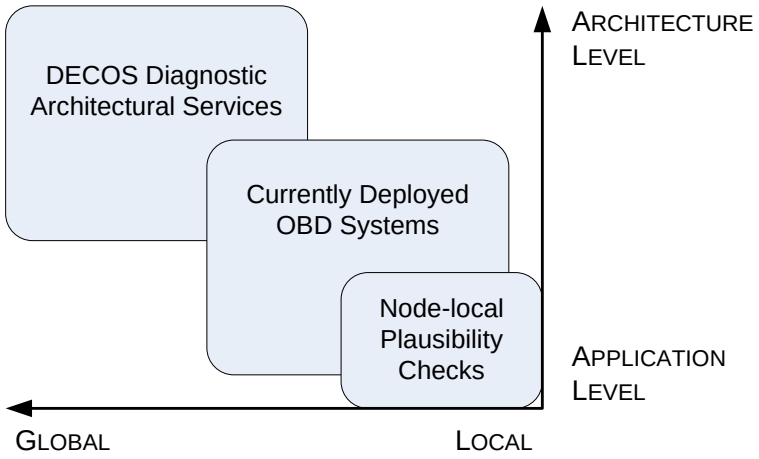

Figure 5.2: Classification of Diagnosis

analysis for this model is presented in Section 5.4. The concept of an Out-of-Norm Assertion (ONA) as the primary diagnostic mechanism is introduced in Section 5.5. For the transport of diagnostic messages a so-called virtual diagnostic network is used that is scope of Section 5.6. The specification and execution of ONAs is presented in Section 5.7. The realization of the detection techniques at component level is discussed in Section 5.8. Finally, the determination of the correct maintenance action is elaborated on in Section 5.9.

5.3 The Maintenance-Oriented Fault Model

The purpose of a model is to develop a reduced representation of the world, that helps in understanding the problem domain [Kop97]. Related fault models presented in [Cri91, Lap92, Pow92, WLS97] are developed for the purpose of fault tolerance. A fault model for maintenance, however, aims at allowing a determination whether a particular fault affecting the system will require a replacement of a FRU. Thus, a fault classification is necessary, that allows deducing the adequate maintenance strategy from tracing back the fault-error-failure chain. In case of integrated architectures such a fault classification needs to include both component hardware and software module faults, since in integrated architectures a component is shared among multiple software modules.

In the following we elaborate on the constituting elements of the maintenance-oriented fault model we consider important in the context of the DECOS architecture.

5.3.1 Unit of Replacement

There exists a strong relationship in the DECOS architecture between the fault hypothesis (i.e. the fault model for fault-tolerance aspects) and the fault model for maintenance. While in the fault hypothesis the FCRs are identified, i.e. the hardware

units limiting the immediate impact of a fault to the system [LH94] of the system, in the maintenance-oriented fault model the FRUs with respect to hardware faults are stated. Typically, there will be a congruence, since maintenance of faulty components is also a key aspect in establishing the required level of dependability in a faul-tolerant system.

Hence, in the DECOS architecture we consider a component as the FCR/FRU for hardware faults and a job as the FCR/FRU for software design faults.

5.3.2 Fault Classification

In the fault hypothesis the statements about the faults at system level are defined that may occur if a FCR exhibits a failure (system level). In case of a maintenance-oriented fault model on the other hand, a classification of the faults affecting a FRU needs to be specified (FRU level). Thus, the two models classify faults at different levels in the fault-error-failure chain [Lap92] as illustrated in 5.3.

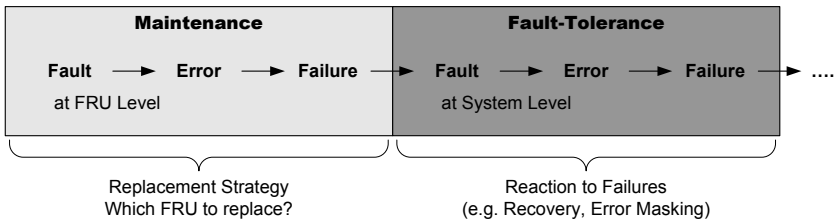

Figure 5.3: The Fault-Error-Failure Chain

As stated in [ALR01] the concept of fault is introduced to stop the recursion of the "fault-error-failure" chain. From a maintenance point of view, we are only interested in categorizing the type of fault of the experienced failure into classes that allow a determination whether a replacement is the correct maintenance strategy. Thus, by reversing the fault-error-failure chain [Lap92], it must be possible for the diagnostic subsystem to determine whether a change of a FRU can eliminate the experienced problem, or if a replacement (i.e. change of hardware or update of software) will prove to be ineffective. On the basis of the maintenance-oriented fault model a corresponding maintenance action for each fault class needs to be stated.

Consequently, we stop the recursion at FRU level. In the context of the DECOS architecture in case of hardware faults the FRU is considered to be complete node computer, while for software faults the FRU is considered to be a job. The fault classification for each FRU needs to be derived by analyzing the prevalent types of faults affecting the given FRU.

Consider for instance a crack in a PCB. Such a crack may originate from wear-out of the material due to environmental (external) stress, such as vibration (e.g., rough roads), shock (e.g., chuckholes, hard landings) and changes in temperature (i.e. expansion and contraction). Depending on operational conditions this crack may cause the component to fail transiently. From a maintenance point of view (at the service

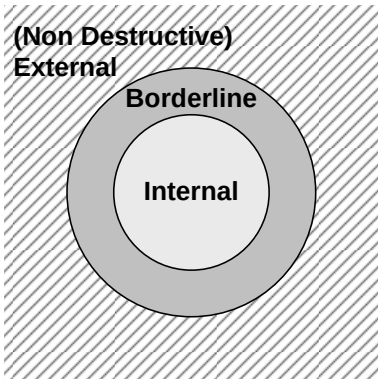

Figure 5.4: Component Fault Model

station) the first level cause due to mechanical stress is not of interest. In analogy the exact element of the FRU that is subject to failure is of limited interest for a service technician. By taking a maintenance-oriented view the most important fact we are interested in is that the hardware fault can only be eliminated by replacement of the FRU. The analysis, which part of the FRU caused the malfunction is in the scope of the inspection of faulty nodes at the OEM (and not part of the maintenance action at the service station).

5.3.3 The Component Fault Model

The model takes the component-based nature of today's distributed systems into account by considering a component as a FRU for hardware faults. Consequently, we devise the following fault classes as illustrated in Figure 5.4.

Faults that originate outside the component boundaries are denoted as *external faults*. We discriminate between *non destructive* and *destructive* external faults. Effects of non destructive external faults can be eliminated by restarting the affected component and applying subsequent state synchronization. Thus, non destructive external faults are characterized by having no permanent effect on the functionality of the component. An example for an such a non-destructive external fault is EMI [KWS00]. By contrast, destructive external faults are characterized by the fact that even after a restart the state of the component remains corrupted (e.g., physically damaged ECU after a car crash). So-called *borderline faults* are the class of faults that cannot be judged to be external or internal with respect to the component boundary. An example for such a fault is a connector fault (a connector consist of two parts, one attached to the component, the other attached to the cable loom). Since this class is responsible for a significant number of system failures [SMM00], we extend the boundary classification of faults as introduced by Laprie [ALR01] by adding the class of borderline faults. Finally, *internal faults* cover those faults that originate from within the FRU boundary (e.g., crack in the PCB). In contrast to external faults, these faults can only be eliminated by a replacement of the component.

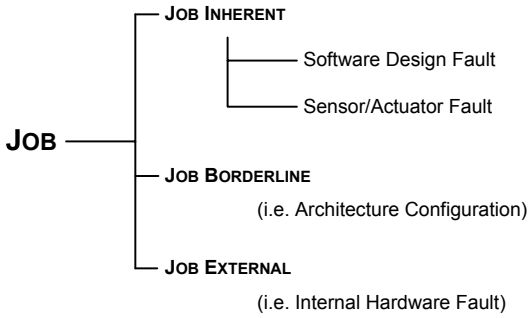

Figure 5.5: Job Fault Model

In the remainder of this thesis we refer to external faults as being non destructive, since from a maintenance perspective, destructive external faults can always be classified as internal by following the fault-error-failure chain.

5.3.4 The Job Fault Model

In the context of integrated architectures, such as the DECOS architecture, a further differentiation of component internal faults is possible. While in architectures with the "1 Function - 1 ECU" design philosophy such a differentiation is futile, in integrated architectures such a finer granularity is important for the discrimination between software faults and hardware faults.

The increasing complexity of software deployed in embedded systems requires the online diagnostic services of an architecture to provide means for identifying software design faults. An empirical study, although based on field data from the telecommunication industry, identified that only a small number of software modules is causing the majority of software related failures during operation [FO00]. If we act on the assumption, that a similar distribution of software faults is also feasible for the automotive/avionic domain, then a correlation of field data gathered by the online diagnostic services of a representative population provides a solid foundation for the identification of software design faults.

As depicted in Figure 5.5 we discriminate for each job between job inherent, job borderline, and job external faults. *Job external* faults are faults affecting the internals of a component but do not origin within the boundaries of the job. In case multiple job external faults can be observed in one component, a component internal hardware fault can be assumed. Similar to the borderline faults at component level, *job borderline* faults are faults affecting the connectors, i.e. the ports of the jobs. Consider for example event-triggered jobs and ports accordingly. In case the jobs are operating as specified in term of sending messages according to an a priori defined probability distribution, there still might be the case where queue overflows occur (i.e. messages are lost). In this case a false configuration of the respective virtual network service is causing system malfunction. Job borderline faults are thus configuration faults. Finally, the class of *job inherent* faults are those faults that

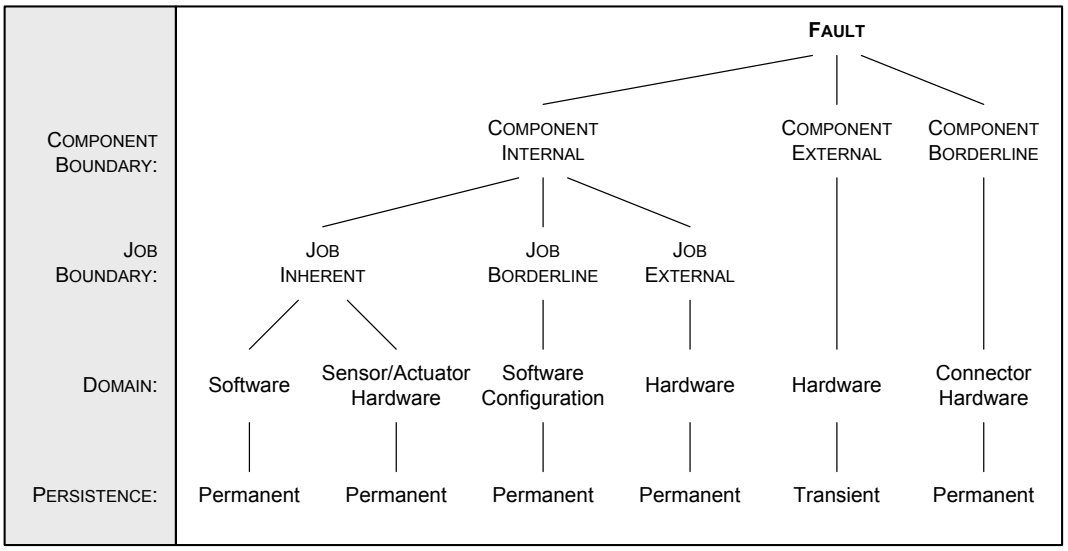

Figure 5.6: Overview of the Maintenance-Oriented Fault Model

are originating from within the job. The class of job inherent faults can be further decomposed into *software design faults* and *sensor/actuator faults*. In the DECOS architecture each job is considered to have exclusive access to its sensors and actuators. Since, in general, one cannot differentiate by observing only the interface state whether a malfunction of the I/O hardware or a software design faults is causing unspecified job behavior, a differentiation of these two types is only possible by including job internal information into the assessment process.

Figure 5.6 gives an overview of the introduced maintenance-oriented fault model and relates the introduced fault model to the terms introduced in [Lap92, ALR01]. The system boundaries are refined into component and job boundaries as the FRUs for hardware and software faults.

5.3.5 Assumptions behind the Fault Model

In order to allow the online diagnostic mechanisms to determine the type of fault that is affecting a particular FRU it is important to make quantitative statements about the underlying failure rates for both transient and permanent faults. By taking into account the studies and numbers presented in Section 3.5 the following assumptions are made for the DECOS maintenance-oriented fault model:

- **Transient Hardware Failure Rate.** The transient failure rate of a FRU with respect to hardware faults is assumed to be in the order of 100.000 FIT, i.e. about 1 year.

- **Duration of Transient Hardware Failures.** The duration of a transient hardware FRU failure can be assumed to be in the order of tens of milliseconds.

For example in [HT98], the transient outage-time of an automotive steering system can be estimated as less than 50 ms.

The time-triggered core physical architecture ensures that transient failures longer than the length of a slot of the Time Division Multiple Access (TDMA) round can be detected by other FRUs.

In current automotive OBD systems, transient failures that are lasting for more than 500 ms are recorded. Failures with a significant shorter duration cannot be detected,

- **Duration of Correlated Transient Hardware Failures.** Correlated FRU failures, i.e. a fault affecting more than one FRU at the same time, are assumed to be experienced within a bounded interval of time. According to the ISO 7637 standard [ISO95] the duration of an EMI burst is in the order of 10 ms.

- **Wearout Indicator.** In the DECOS model we regard increase of transient failures of a FRU as a suitable indicator for wearout of the electronic device [Con02].

- **Permanent Hardware Failure Rate.** The permanent failure rate of a FRU with respect to hardware faults is considered to be in the order of 100 FIT, i.e. about 1000 years [PMH98].

- **Software Faults Distribution.** We assume that safety-critical jobs are certified to the necessary degree and thus free of software design faults. In case of non safety-critical jobs, we assume that a minority of the deployed software FRUs is causing the majority of software related failures during operation [FO00].

5.4 Suitability Analysis of the Maintenance-Oriented Fault Model

The following section is devoted to underpinning the suitability of the introduced maintenance-oriented fault model to differentiate faults experienced in real-world systems. We present an analysis that covers examples from literature for both the component and job classification.

5.4.1 Component Fault Model

Internal

In the following we will discuss component internal fault sources on the basis of representative examples found in literature.

Printed Circuit Board The PCB interconnects the constituting hardware elements of a node.

- **Design.** A typical design fault with respect to PCB is an erroneous layout. Consider for example faulty spacing between wires or incorrect placing of electrical elements on the PCB.
- **Manufacturing.** Manufacturing faults include all faults originating from the technical process of creating the PCB. Here, solder mask problems, defective vias or wrong assembly are among the typical faults. Also soldering related faults, such as defective solder connections, shorts or loose contacts fall into this category.
- **Operational.** Environmental stress factors (e.g., vibration, shock, humidity, chemicals) can lead to subsequent failure of electronic devices. Continuous exposure to these factors can result in cracks in the PCB due to wear-out of the board or over-stress resulting from thermal cycling or shock. The PCB is the primary source of component failure in case the ECU is exposed to stress conditions [WWS99].

Discrete Elements According to field studies [PMH98, WWS99] resistors diodes and transistors have the lowest failure rates. Capacitors are an exception since these elements are more affected by aging processes and thus having a higher failure rate due to wearout that other discrete electronic elements.

Quartz Since measurement of time in computer systems is based on frequency of oscillation of a quartz crystal, the correct functionality of these elements is of paramount importance. In case the system relies on precise timing information, environmental influences and wearout can have a significant impact on the drift of the clock. Consider for example systems where components are synchronized in order to achieve a consistent global timing information. Here, a defective quartz can cause a component failure due to loss of synchronization. As indicated in [WWS99] the failure rate cannot be neglected. During operation the quartz can be influenced by many factors. Low power supply, thermal cycling and mechanical damage due to shock and vibration are probably the most common causes for permanent or transient faults [Sch95].

Integrated Circuits As indicated by the failure rates provided in [PMH98] the higher the integration of the electronic elements, the higher the likelihood of failure. Therefore, integrated circuits are causing a significant number of component failures. In the following we will identify possible design, manufacturing and operational faults. An excellent overview on semiconductor faults can be found in [GAM+02].

- **Design.** The reliability of electronic devices was subject to significant improvements in the last decades. However, the downsizing of semiconductor features

has lead to decrease in the gate oxide thicknesses and the distance between metallization, resulting in higher electric fields across the gate and possibility of failure, such as gate oxide breakdown and hot carrier damage [MPG02].

Due to the increasing level of complexity of modern integrated circuits the likelihood of design faults is non-negligible. For example Lev and Chao [LC02] state that, in nanometer design, wiring delay accounts for the vast majority of overall delay. The downsizing causes that the delay is shifting from gates to wires. By 90 nm, wiring delay will account for some 75% of the overall delay. Thus, the shrinking of geometries in semiconductor design has significant impact on future design processes. Besides wiring delay, cross coupling effects originating from incorrect design can result in transient failures during operation.

- **Manufacturing.** Due to semiconductor process variations such as intra-chip variances or mask alignment and manufacturing residuals the likelihood of reoccurring permanent faults leading to transient failures is growing [Con02]. For example consider a short of metal lines caused by an unexposed photo resist or a solid-state particle deposited on the metal layer before metal lithography.

  Failures of system components caused by these effects of manufacturing shortcomings can cause significant trouble for the maintenance engineer. Consider for instance process variations that cause temperature dependent failures (e.g., at windshield wiper control unit). Due to defective mask alignment (adjudged fault) the IC shows temperature dependent failures. Thus, if the temperature lies within the range between $-20°$ and $-10°$ Celsius the chip exhibits unspecified behavior. As soon as the customer brings his car to the service station, the experienced failure cannot be reproduced because of the different temperature. Since process variations can affect only a batch, these faults are hard to identify and and can lead to major maintenance problems. This is also an example of a fault that is made active due to environmental influences.

- **Operational.** According to Constantinescu [Con02] the primary cause for the significant increase of soft error rates are shrinking geometries, lower power voltages and higher frequencies. These result in higher sensitivity to neutron and alpha particles, and consequently have an impact on dependability by increasing the transient failure rates [GAM+02].

  Another significant source of failure is variability in power supply and temperature effects. As stated in [Won99] temperature has strong influence on the properties of semiconductor materials. On the device level, mainly degradation and breakdown of oxides are the main cause of failure.

Borderline

Wiring and connector problems account for a significant proportion of electronic system failures. Several studies document the significant proportion of connector

and wiring failures in distributed embedded systems. Field data cited by Swingler et al. [SMM00] indicate that more than 30% of electrical failures are attributed to connection problems. Considering, that a luxury car can have up to 400 connectors this number underpins the potential for failures due to connectors in the automotive domain. In the avionic domain, Galler and Slenski identify interconnection problems as the major cause of aircraft electrical equipment failures [GS91] with a percentage of 36%. These numbers are underpinned by a study of the US Air Force reporting that 43% of mishaps related to electrical systems were due to connectors and wirings [SS01, For95].

The reliability of the interconnection system is of great importance for the correct operation of an electronic system. The physical interface between the electronic control units (i.e. the components) and the interconnection system remains one of the weakest links in terms of reliability, implying potentially catastrophic consequences [McB93].

According to Tanner [Tan93] several issues concerning the reliability of connectors can be identified:

- Terminals backing out of housing. This frequent problem, both affecting warranty claims and production, can be reduced with the use of secondary locking features.
- Terminals damaged on production line. Bent pins or damaged tabs are the most common form for of terminal damage. These damages can result from harness assembly, transit, or during installation.
- Partially mated connectors. With excessive connector mating force a poorly designed connector latch causes malfunction.
- Breakdown of contact at crimp. Poor crimping of the contact at the manufacturing stage can significantly influence the quality of the interconnection.
- Failure of contact due to operational conditions. In contrast to the hitherto presented problem sources, the last one is a consequence of operational conditions rather than improper design or assembly. Due to environmental stress (e.g., thermal, shock, vibration) the male and female parts of a connector can move relative to each other leading to gradual increase of the contact resistance. This phenomenon is termed *fretting wear* and is defined as wear caused by small repetitive motion in an apparently stationary situation [vDvM96]. Especially in combination with corrosion it is very likely that such a connection is responsible for transient anomalies in the system. Kimseng [KHTP99] for instance notices, that pin and socket type interconnects between the upper and lower housings are particularly vulnerable to small relative displacements due to the vibrations and thermal cycling of the automotive environment.

A significant problem regarding connector and wiring failures is the fact that the failure analysis or the testing procedure may itself be a corrective action for

the source of the failure (e.g., such as when resetting a connector or when applying wipe to a connector electrical pad) [PR92]. The same is pointed out by [KHTP99], who states that analysis and repair is often difficult due to the possibility that any evidence of failure is inadvertently destroyed during extraction or inspection.

External

In the following we will discuss the most important external faults that have an impact on the functionality of the DECOS components. The class of external faults covers external influences such as cosmic radiation, temperature, EMI, shock, vibration, and humidity, only to name a few [IEE99]. An extensive list of environmental factors having impact on the reliability of a component can be found in [DoD98, p. 7-129].

Cosmic Radiation In aerospace transient disturbances of components are frequently caused by Single Event Upsets (SEUs) originating from cosmic radiation. According to [Ram01] such radiation induced failures in avionics are caused by uranium and thorium contaminants, and secondary cosmic rays. Among the consequences of radiation are aging effects (wearout), embrittlement of materials, and overstress soft errors in electronic hardware.

In [Nor96a] Single Event Upsets (SEUs) caused by cosmic rays in avionics are investigated. Based on the experimental evidence from measured in-flight occurrences of SEUs fault rates in dependence of the flight altitude are evaluated and the sources of such incidents identified. However, SEU are not restricted to higher altitudes as shown in [Nor96b].

Electromagnetic Interference EMI imposes a serious threat to the intended function of electronic systems deployed in various application environments. In the following we will give a short overview of EMI related problems in the avionic and automotive domain.

One major source of EMI in the avionic domain is the effect of lightning on aircrafts. Besides severe effects on the aircraft skin (e.g., melting, deformation due to pressure waves), damage in externally mounted materials and vaporization of conductors, a serious consequence of lightning for the electronic equipment are the electric and magnetic fields. In [Pod90] a 16.5% failure rate of electronic equipment of commercial airlines due to lightning strokes is reported. Since modern aircraft highly depend on the correct functionality of the electronic flight control system standards exist, to provide necessary aircraft protection [Pod90].

According to Ladkin [Lad97] there are no reports so far of interference with electronic flight control on the Airbus 320/330/340 or the Boeing B777 caused by EMI. However, in military aircraft some incidents are documented. For instance, five crashes of Blackhawks helicopters shortly after their introduction into service

were found to be due to EMI from very strong radar and radio transmitters with the electronic flight control systems [Lad97].

An additional problem in avionics is the increasing numbers and use of Portable Electronic Devices (PEDs) onboard an airplane [LXR+02]. However, Boeing has not been able to find a definite correlation between PEDs and the reported airplane anomalies [Don00].

Similarly, in the automotive domain the increasing use of electronics makes cars more susceptible to problems originating from EMI, and thus makes it a major consideration in vehicle electrical system design [Nob94]. For example in [Ber02] serious effects of EMI are mentioned, namely the unexpected shut off of car engines on highway overpasses. Another example for transient disturbances generated by EMI is noise of the ignition system of a car [Nob92]. The UK based Motor Industry Software Reliability Association (MISRA) consortium has released guidelines for dealing with EMI in automotive environments [MNW98]. The guidelines consider interference effects on various aspects of processing, for example communication lines, digital and analogue inputs, corruption of memory and loss of control of the processor. Similar impacts of EMI have been studied in [PKOP99]. An interesting study of EMI levels in urban areas can be found [GSOS01]. The measurement results are compared against the requirements of the automotive industries in order to ensure correct functionality of on-board electronics.

A study on effects of electromagnetic interference on controller-computer upsets and system stability revealed that controllers are more susceptible to these unwanted noises due to shrinking device size, lower switching energy, and higher speed operations [KWS00]. Typically, these external faults are likely to be transient and cause primary functional error modes in digital systems.

Environmental Stress Factors Transport vehicles, such as cars and airplanes, are usually exposed to harsh environmental conditions [Fle01]. Especially, climatic and mechanical stress decreases the lifetime of electronic equipment. For example, humidity, extreme temperature and moisture in combination with stress factors such as shock and vibration results in increasing wearout rates [WBC00, WWS99].

Just to give an impression, in the automotive industry temperatures can reach up to 200°C on the engine or even 800°C at the exhaustion system. Similar, the vibration and shock levels can reach up to 50g [Won99]. For example in the cabin the vibration is usually in the magnitude of 10g, under the hood 20g and wheel or engine mounted devices experience up to 40g.

Thermal cycling, continuous vibration and shocks and environmental conditions like salt spray, dust or gravel that weaken the protection mechanisms (e.g., sealing, housing) are a serious threat to the reliability of of electronic devices deployed in electronic architectures.

Due to continuous exposure to environmental stress, external faults can be transformed into internal component hardware faults (e.g., continuous shock causes crack

in PCB). Such an accumulation of incremental damage beyond the endurance of the material is termed *wear-out* fault [Ram01].

Wiring Wiring related problems are posing a serious threat to safety-critical systems. It is an acknowledged fact that every densely wired system is vulnerable to consequences of wiring problems. For instance Swissair 111 and TWA 800 have crashed because of faulty wiring [FH01].

According to [SS01] the aging process of wiring can be understood as degraded performance due to accumulated damage from long-term exposure. This includes damages resulting primarily from operational conditions, such as damages from chemical, thermal, electrical, and mechanical stress. Besides these stresses induced by the operational environment damages also originate from installation and maintenance practices. Such wiring failures frequently appear as broken conductors and damaged insulation which can be disrupt electrical signals and/or lead to arcing, that may have fatal consequences.

One approach of improving the safety of today's airplanes is to provide mechanisms that allow a better detection of the tiny anomalies caused by defective cabling. These diagnostic mechanisms need to monitor the wires not only during maintenance but rather during the operation of the plane. According to [FH01] a single arc fault may last only 1.25 ms, and a series of events may last 20-30 ms. However, these spurious failures must be distinguished from transient failures induced by the environment (e.g., jet engine ignition).

5.4.2 Job Fault Model

Inherent

The class of job inherent faults as introduced in 5.3.4 is divided into software faults and transducer (i.e. sensors and actuators) faults. In the following we will discuss why such a classification is feasible for today's distributed embedded real-time systems.

Software Faults Although automatic code generation tools such as TargetLink from dSpace or the Real-Time Workshop for MATLAB/Simulink are increasingly becoming accepted in industry [WP99] in order to reduce software implementation faults [MIS98] and speed up development, the increasing complexity of applications leads to an increased probability of software design faults. In particular, Heisenbugs remain frequently undetected and can only be identified by a fleet analysis during full operation of the product. For example, a software bug in an electronic management unit of the fuel pump caused some cars to stall if the fuel tank was below one third full. The resulting recalls not only impose a serious financial burden for the manufacturer but also have a significant impact on the reputation of the products.

In [Web92] the support of integrated diagnostics for software is underpinned by the provision of statistics indicating that 17% of the efforts associated with software

maintenance are for correcting faults. Furthermore, 54% of the efforts associated with software support require an integrated set of diagnostic tools and techniques. The importance for the detection of software faults during system operation is stressed by stating the fact that despite all efforts to reduce software faults during development, there will still be latent software faults during testing and deployment (at least for non safety-critical systems). The issue of faults that can only be identified during operation is also raised in [GW02] in the context of the automotive domain. During product development and testing low quality issues are relatively easy to identify because they are uncovered with smaller sample size. The problem of current vehicle testing process is the identification of statistically very unlikely occurring incidents that become only identifiable after high volume production.

A recent study of software faults revealed that only a small number of software modules contain most of the faults discovered during pre-release testing [FO00] supporting the results of [Ada84]. However, the discovery of these faults during pre-release testing is a very challenging task. In case of software faults detected during operation a distribution according to the *20-80 rule* has been identified, indicating that 20% of the software modules are causing 80% of the software related failures during operation.

Transducer Faults Sensors and actuators are the linkage between the controlling computer system and the controlled object. In the DECOS architecture each job is considered to have exclusive access to its sensors and actuators (e.g., electromechanical brake, window lifter, wheel speed sensor). An overview of sensors currently deployed in automotive industry can be found in [Fle01, Bos02]. For the avionics domain the reader is referred to [MS03]. In the automotive domain the expected lifetime of sensors is assumed to be in the order of the lifetime of the car. For example in [Par01] the lifetime of automotive pressure sensors is specified between 10 and 15 years.

One approach to the highly application-specific diagnosis of job inherent faults is model-based diagnosis [PW03]. Based on a diagnostic model the application programmer transforms a model into application-specific assertions that are checked at run time. In [Nyb02] an example for model-based diagnosis in the automotive domain is presented. The author presents a diagnosis solution for the air-intake system of an automotive engine.

Borderline

The configuration of a distributed embedded real-time system is typically tool supported in order to minimize the possibility of faulty configurations (for instance see [Pol04]). The class of borderline faults comprises those faults that emerge by deriving the configuration parameters on the basis of a communication model that is based on assumptions that do not hold in reality.

Consider for example an event-triggered legacy application. Temporal correctness of such an application can depend on temporal properties, such as bandwidth guarantees, bounds on communication latencies, and predefined message orderings. Furthermore, knowledge about the temporal behavior of communication activities is essential for the dimensioning of message buffers as required to tolerate temporary imbalances of message interarrival and service times [Kle75]. If a subset of the assumptions of a legacy subsystem was made implicitly and not described in technical documentation, then determining a valid configuration is complicated. With incomplete knowledge about the assumptions that have been made by legacy applications concerning the underlying architecture, finding a consistent configuration becomes a non-trivial and error-prone activity. We denote any misconfiguration of the architectural services that results from incomplete knowledge about legacy applications as a borderline fault.

External

A job external fault can be mapped onto a component internal hardware fault. In case of a one-to-one mapping between jobs and components as in federated systems, this differentiation is obsolete. However, in the context of an integrated architecture such a differentiation is important to determine whether a component internal fault is a job inherent fault.

5.5 Out-of-Norm Assertions

In the following section we will introduce Out-of-Norm Assertions (ONAs) as a generic mechanism operating on the consistent distributed state induced by a sparse time base for the encoding of fault patterns in the value, time and space domain. Besides supporting the detection of a violation of a component's service specification, ONAs establish means for the detection of system irregularities that cannot be forced into a bivalent assessment scheme at the time of occurrence. ONAs correlate spurious states in value, space and time in order to allow component assessment.

5.5.1 Distributed State

This section discusses the notion of state in distributed systems and the consequences of faults on the distributed state. We start by elaborating on the component state and express the role of the interface state as the part of the component state that is visible to all other components of the system. Furthermore, we exploit the concept of the sparse time base in order to reach a consistent distributed state. Finally, we introduce the concept of fault patterns on the distributed state to model and analyze the effects of faults. The concept of ONA is based on the notion of state as introduced in Section 2.2.

Component State

The definition of a system component as introduced in Section 2.3, i.e. the inseparability of the software from its executing hardware, provides the ability to observe and describe the behavior of the component in relation to physical time. A component possesses a physical clock, the ticks of which trigger the progression from one processing step to the successive one. As a result of each processing step, a component will deterministically update its internal variables – based on the program code, the inputs, and the values of the internal variables before the processing step. We denote this component as *time-aware*, since the underlying conceptual model of a component incorporates the notion of physical time.

At startup, a component has not processed any inputs and the values of the internal variables of the component equal the initialization values. Consequently, at this point in time, the future outputs of a deterministic component depend solely on the program code and the initialization values. These static data structures of a component are usually denoted as the *initialization state*. The *history state*, on the other hand, incorporates the dynamic data structures that change over time [Kop97]. If we do not wish to perform this differentiation, we simply speak of the *state* of the component at a particular point in time.

In distributed computer systems, the components interact by the exchange of messages across service interfaces to realize emerging services. Based on the analysis of the interactions between a component and its environment a strict separation of concerns at the interface level [RX97] helps to reduce complexity by enforcing a more structured design. A service interface that is provided to link components together is called a LIF [KS03a]. One can further distinguish between the Service Providing Linking Interface (SPLIF) and the Service Requesting Linking Interface (SRLIF) of a real-time communication LIF. While the SPLIF offers the service of the component to all other components of the distributed system, the SRLIF requests services from other components.

Formally, we can formulate the component state $S_{\text{component}}$ of a component A at time t as following:

$$S^A_{\text{component}}(t) = \left(\bigcup_i S^{A^i}_{\text{interface}}(t) \right) \cup S^A_{\text{internal}}(t) \tag{5.1}$$

where $S^{A^i}_{\text{interface}}(t)$ denotes the interface state of component A (either an SPLIF or an SRLIF) at time t with respect to interface $i, i \in \{1, 2, \ldots, n\}$ and $n \in \mathbb{N}$. The internal state $S^A_{\text{internal}}(t)$ of component A denotes the part of the state that is not made explicit through interfaces (e.g., temporary results, registers, program counter).

Interface State

Components are interconnected with other components by means of exchanging messages across LIFs to realize emerging services. LIFs can be specified at two levels: the

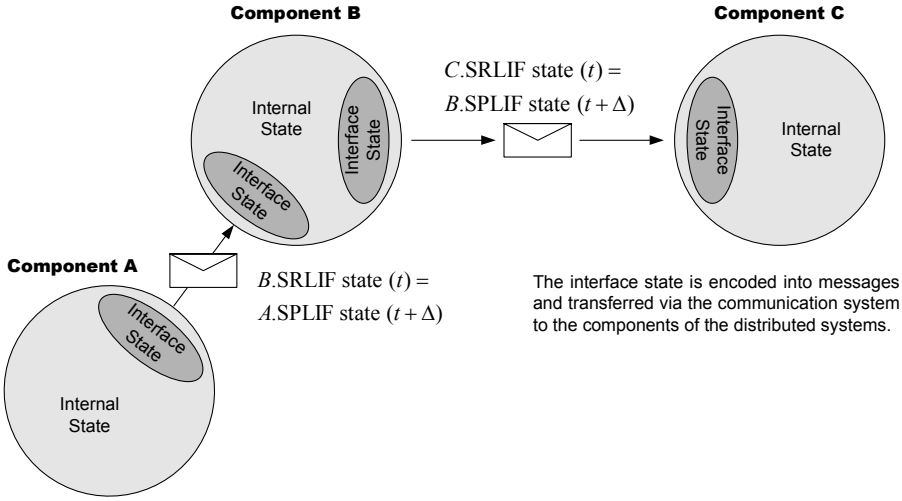

Figure 5.7: Interface State

operational level incorporates syntactic and temporal aspects, while the meta-level denotes the semantics [KS03a]:

1. **Syntactic Specification.** The syntactic specification defines the structure and name of the data elements exchanged via the interface. Thus, the concept of syntax is used to construct structured information from basic information units.

2. **Temporal Specification.** The temporal specification determines the temporal sequence of message exchanges.

3. **Meta-Level Specification.** The semantic specification assigns a meaning to the structured information.

The specification of LIFs is a statement about the messages that arrive at SRLIFs and depart from SPLIFs. If the messages produced by a component comply to its specification at the two levels, a component is correct. Otherwise, the component exhibits a message failure and the component is denoted as faulty. Associated with such a message failure is an incorrect *interface state*, which is the state of a component as viewed from a particular interface [GIJ+02]. In contrast to the internal state, the *interface state is accessible* to other components in the distributed system. As depicted in Figure 5.7 the interface state of each component is encoded into messages and transferred via the communication system. Thereby a replication of the interface state is performed. Note, that the internal component state remains hidden.

- **Interface State of SPLIFs.** While intermediate computational results (in general) only effect the internal state variables (i.e. the internal state) of a component, the final computational results are made available via the SPLIFs. Thus, the information from private, internal state variables is propagated into

the component's interface state containing the set of state variables that are explicitly exported by a component allowing access by other components. In other words, the interface state of a component are those state variables that are transported to other components by exploiting the communication service.

- **Interface State of SRLIFs.** By exploiting the SRLIF, a component adapts its state variables with the state variables of the sending component. This way, a component acquires access to foreign state, which functions as input for subsequent computations – in the same way as the history of the component. By processing the interface state of an SRLIF the synchronized state of the sending component is propagated to the state of the receiving component and thus determines future behavior of the component provided via the SPLIF.

Due to the importance of the interface state concept, we also introduce the notion of *visible component state*, which consists solely of the interface state of a component (i.e. omitting the internal state). Formally, the visible component state $S^A_{\text{component,visible}}(t)$ of a component A at time t is defined as follows:

$$S^A_{\text{component,visible}}(t) = \left(\bigcup_i S^{A^i}_{\text{interface}}(t) \right) \tag{5.2}$$

Distributed State

If the time base in a distributed system is dense (the events are allowed to occur at any instant of the timeline), then it is in general not possible to generate a consistent temporal order on the basis of the time-stamps [Kop97]. Due to the impossibility of synchronizing clocks perfectly and the denseness property of real time, there is always the possibility that a single event is time-stamped by two clocks with a difference of one tick. By introducing the concept of a *sparse time base* the ordering of events can be restored without execution of agreement protocols only based on timestamps [Kop92]. In the sparse time model the continuum of time is partitioned into an infinite sequence of alternating durations of activity (π) and silence (Δ). Thereby, the occurrence of significant events is restricted to the activity intervals of a globally synchronized action lattice.

The activity intervals of the sparse time base form a synchronized system-wide action lattice. The interval of silence (Δ) on the sparse time base is a system wide consistent dividing line between the past and the future and the interval when the state of the distributed system can be defined. This consistent dividing line between the past and the future is illustrated in Figure 5.8.

At any point in time t, where t is an instant on the sparse time base corresponding to an interval of silence Δ, the union of the visible state of each component is denoted as the *distributed state*. During the interval of activity π the distributed state is undefined. The state of a distributed system $S_{\text{distributed}}(t)$ at time t can thus be

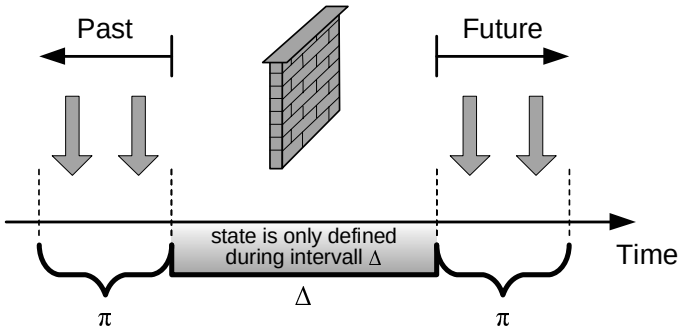

Figure 5.8: Sparse Time Base

devised as:

$$S_{\text{distributed}}(t) = \left(\bigcup_{A} S^{A}_{\text{component,visible}}(t) \right) \tag{5.3}$$

where $S^{A}_{\text{component,visible}}(t)$ denotes the visible state of a component A at time t.

Adversely, the component state $S^{A}_{\text{component,visible}}(t)$ in a component A of the distributed system includes part of the distributed state $S_{\text{distributed}}(t)$. Through the exchange of information via the communication system, in each component a local manifestation of a part of the distributed state occurs. The part depends on the communication relationships between the components. This local manifestation of the distributed state will be exploited in Section 5.5.2 for capturing certain types of diagnostic evidence, which function as an indicator for the occurrence of faults.

Consequences of Faults on the Distributed State

When a fault hits one or more constituting parts of the distributed system, a change of state can occur that leads to an unintended state denoted as an error [Lap92]. Depending on the type of fault (e.g., internal or external fault, software or hardware fault), the unintended state will exhibit a characteristic manifestation in time, value and space. To capture the characteristics of the fault-induced distributed state changes, we introduce the concept of *fault pattern*. A fault pattern is the set of state variables that has been identified as subject to fault-induced state changes along with corresponding properties in value, space, and time. Different types of faults show different fault patterns on the distributed state.

- **Value Dimension.** The value dimension denotes a subset of the value domain of the selected state variables. This subset of the value domain is characteristic to the fault covered by the fault pattern.
- **Time Dimension.** The time dimension covers the persistence of the change in state. The time dimension makes statements about points in time, the durations, and the variability of the points in time and the variability of the durations of fault-induced state changes. Additional indicators on the time

Dimension	Fault Patterns		
	Wearout	Massive Transients	Connector Fault
Time	increasing frequency as time progresses	approximately at the same time (within a small delta)	arbitrary
Space	one component only	multiple components with spatial proximity	one component only
Value	increasing deviation from correct value, at the verge of becoming incorrect	multiple bit flips	message omissions on a channel

Figure 5.9: Summarized Fault Patterns

dimension are the time between successive fault-induced state changes and the frequency of state changes.

- **Space Dimension.** The space dimension specifies the locality and spatial expansion of the observable state changes. For example, the space dimension can encode whether a fault affects a single or multiple components and the physical proximity of the components hosting the changed state variables.

An important input for deriving fault patterns is the fault hypothesis. The fault hypothesis expresses the statements about the considered FCRs, the failure modes, the temporal properties, the failure frequency and the error detection latency [Kop04]. In case the fault hypothesis states that a FCR with respect to software and hardware faults is a component, then a distinction between hardware and software faults is only possible by including the spatial dimension into the fault patterns.

Another example may be a fault hypothesis that states that the software has to be free of design faults (as required in some safety-critical systems). In this case only hardware faults need to be considered in case of component failure.

In the following we exemplify some typical fault patterns, which are summarized in Figure 5.9.

- **Wearout fault pattern:** A *wearout failure* is defined as a failure due to monotonic accumulation of incremental damage beyond the endurance of the material [Ram01]. Such a wearout failure will exhibit a fault pattern typical for intermittent type faults [Lap92]. This type of fault affects only a single component (space dimension) and reoccurs repeatedly at the same location at higher rates with decreasing intervals [Con02].
- **Massive transient disturbances fault pattern:** *Massive transient disturbances* (e.g., due to EMI) are an example for the class of faults typically affecting multiple components at the same time. EMI causes correlated effects on the entire system that usually cause no physical damage to hardware [KWS00].
- **Connector Fault:** A *connector failure* may occur at an arbitrary point in time. Characteristic for the connector fault is the space and value dimension. A connector fault is assumed to affect only one component at a time and manifests itself as a message omission on a channel.

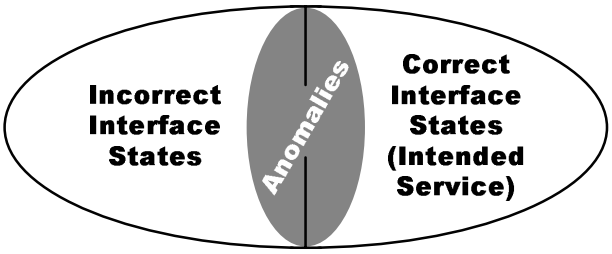

Figure 5.10: State Space

Deriving Fault Patterns

Typically, defective control units are returned to the OEM for warranty analysis [AM02]. Although this off-line analysis is not in the scope of the DECOS integrated diagnostic architecture, the information gained through off-line analysis has a major impact on the design of the fault patterns. An optimization of the patterns in order to support the identification of those faults that have been identified to cause the majority of failures is of paramount importance to the effectiveness of the deployed diagnostic mechanisms. Studies of faults affecting ECUs used in automotive applications underpin the so-called Pareto-principle, i.e. a phenomenon that can have many theoretical causes has in reality only a few [PMH98].

In order to derive the fault patterns for prevalent fault types causing the majority of system failures, a thorough analysis of field data (provided by industry) and fault injection techniques is necessary. Statistics on the types of faults affecting products in operation will help to derive those fault patterns that will help to identify the faults that will most likely affect the system (e.g., car, aircraft) during operation.

5.5.2 Definition of Out-of-Norm Assertions

The specification of a component describes the services that are offered via the LIFs to other components. Thus, it is possible to decide whether a component adheres to its specification or exhibits a behavior deviating from the intended service. In general the specification is a statement about services not about state. Since the provision of a correct service by a component implies a correct SPLIF interface state, the behavior of a component with respect to its service specification can only be checked by continuously evaluating the interface state. An interface state error is thus an indication of an upcoming failure of the respective service of a component.

In contrast to an interface state error, that is a definitive violation of the specification – since an interface state error corresponds to a component failure – an *anomaly* is an interface state that can only be judged as correct or incorrect at time of occurrence by the omniscient observer, but not by the computer system (e.g., due to missing a priori knowledge or redundancy to decide on the correctness). Only by continuously evaluating such anomalies over time, a (definite) classification of the experienced behavior, i.e. a judgement whether the intended service is provided, can

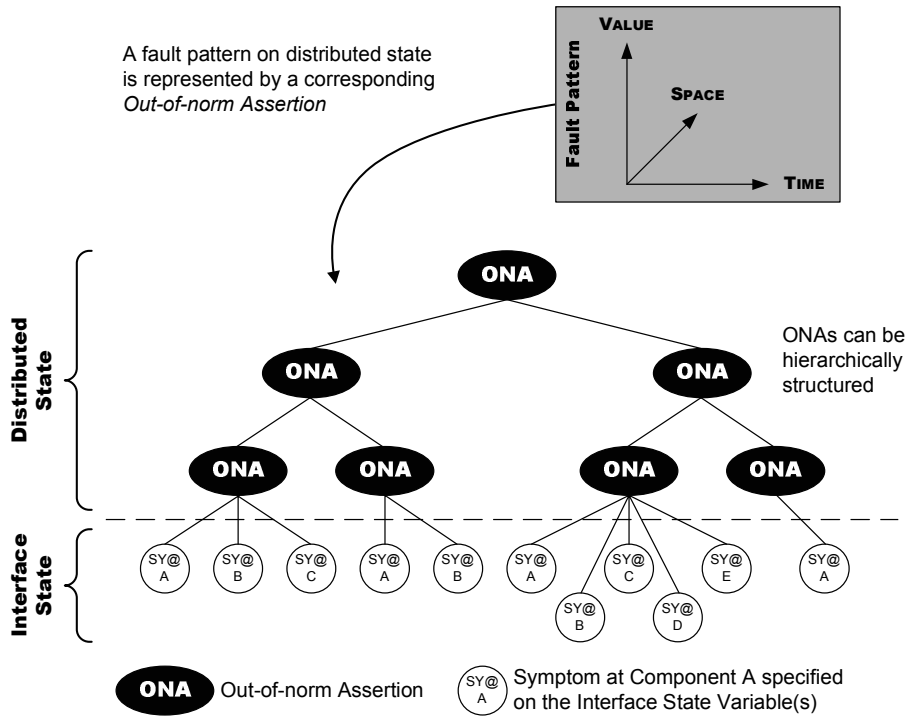

Figure 5.11: Definition of Out-of-Norm Assertion

be derived. Figure 5.10 shows this consideration. While incorrect interface states are perceived as failures of the respective component, anomalies are a deviation from the expected, regular interface states and require further assessment over time.

We define an *Out-of-Norm Assertion (ONA)* as a predicate on the distributed system state that encodes a fault pattern in the value, time and space domain as introduced in Section 5.5.1. ONAs are deterministically triggered, whenever all symptoms of a particular fault pattern are detected on the distributed state. A *symptom* is a set of interface state variables of a particular component that are monitored to detect deviations from the LIF specification. An ONA will likely be composed of more than one symptom, each operating on the interface state of different components. As depicted in Figure 5.11 ONAs can be hierarchically structured. This allows for the exploitation of identified symptoms for the implementation of different ONAs.

In contrast to conventional error detection techniques, ONAs do not provide a definite classification whether a component is correct or incorrect in case only a subset of the specified symptoms fire. In this case, we speak of an anomaly, i.e. we cannot ascribe the behavior of the component to a specific fault pattern. In order to decide on the correctness of a component, an assessment over time is necessary. The repeated evaluation of evidence gathered by ONAs provides the foundation for the analysis process that ultimately decides whether a component is correct. ONAs are gathering evidence in order to decide on a particular pattern affecting the state of the system. This process can be compared with a gathering evidence of different

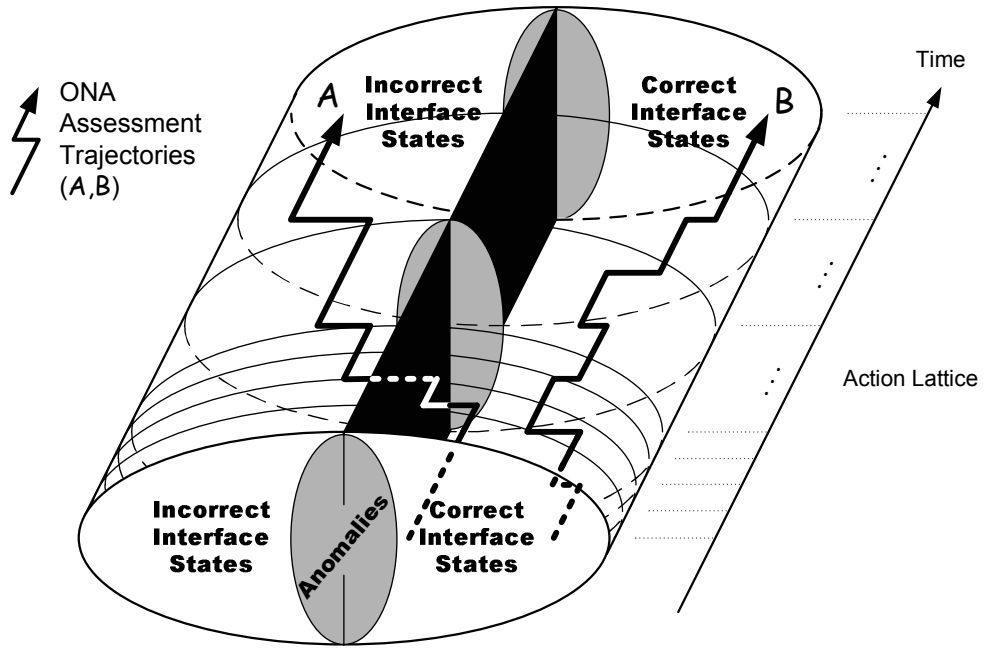

Figure 5.12: Assessment Process

diagnostic techniques in medicine (e.g., temperature measurement, computer tomography, x-ray). In case sufficient evidence is gathered, a suspicion for a particular disease is confirmed or falsified.

Note, although an ONA fires deterministically, i.e. evaluates to either $\mathbb{T}$ or $\mathbb{F}$, the fault pattern encoded in the ONA may only be derived in a probabilistic manner (e.g., from field data). For example, a connector failure can be detected by monitoring the communication channel in a distributed system. In case of replicated channels, a transient failure of one channel can be represented by a connector fault pattern. Since connector failures account for a significant portion of communication faults [SM99], it follows that a message omission failure on one channel is likely due to a connector fault (e.g., loose contacts). However, there will be message omission failures resulting also from other faults, although with a significantly lower probability.

By interpreting the information of all components of the system, correlated failures can be identified that allow to distinguish between external and internal faults. For example, while transient external faults (e.g., due to EMI) randomly effect components, intermittent type faults occur repeatedly at the same component at a higher rate [Con02] (e.g., solder joint cracks). This distinction is especially useful in case of integrated architectures such as IMA [Aer91], where components are shared among multiple applications. Here, the determination of experienced failures to a particular application or to a set of applications is important to decide whether a software or hardware fault is active.

The evaluation process performed by the diagnostic subsystem is illustrated in Figure 5.12. The evaluation process is based on a consistent notion of state, which

is provided through the action lattice of the sparse time base. The arrows in Figure 5.12 indicate the assessment trajectories. At first both arrows show component conformance with the specification, i.e. correct interface states. As time progresses arrow A exhibits an increasing confidence for a violation of the specification, while arrow B indicates a component behavior in accordance with the specified service.

Classification of Symptoms: Self- vs. Cross-Checking

If a component evaluates its own symptoms, we denote this process as *self-checking*. However, when the component is affected by a fault, it cannot be assumed that the error detection mechanisms within this component are unaffected by this fault. Thus, there is always the possibility that the judgement of a faulty component on its correctness is misleading.

In contrast, the interface state is revealed to other components via the exchange of messages through the communication system and can be checked independently by all other nodes. We denote this type of checking of symptoms with respect to specification conformity as *cross-checking*. The validity of such symptoms that can be tested by other components is more trustworthy than the results of checks performed by the node under inspection. This concept conforms to the established architectural principle in safety-critical architectures of mutual error detection [Kop03].

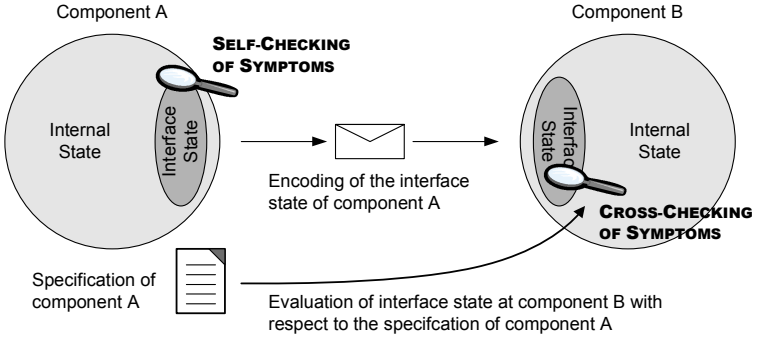

Figure 5.13: Self-Checking vs. Cross-Checking

As stated before, the interface state is transferred to all receiving components via the communication service (i.e. the interface state is encoded into messages), it follows that detection techniques on the interface state at the receiving component are a mechanism to assess the correctness of the sending component. As illustrated in Figure 5.13, component B is able to apply error detection on the interface state with respect to the specification of component A.

Consequently, by applying ONAs we introduce symptoms as *constraints on the interface state* to assess the condition of a system component. Fundamental to this concept is the fact that the interface state of the sending node (Component A) is mapped onto the interface state of the receiving node (Component B) as indicated in Figure 5.14. As depicted, a fault in component A causes an error in the interface state of the LIF. Subsequently, this error causes a message failure (e.g., timing failure,

value failure). Consequently, a failure of the sender manifests itself as an error in the interface state of the receiver (with respect to the specification of the sending node). In case the interface error with respect to the specification of component

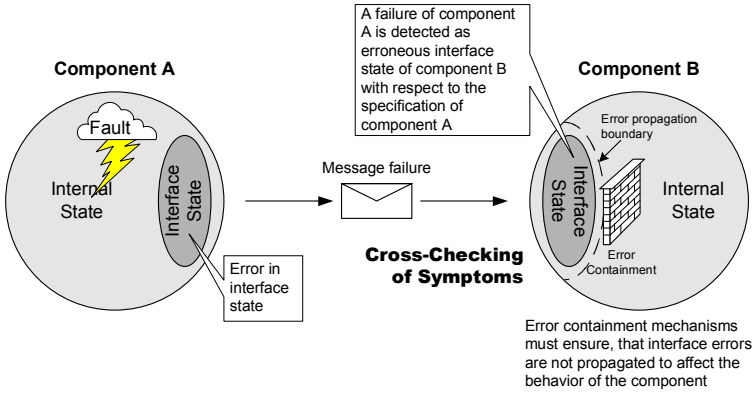

Figure 5.14: A Failure of the Sender is Detected as an Error in the Interface State of the Receiver

A remains undetected, the error propagates to the state of component B and can lead to a consequent failure of component B. For that reason error containment mechanisms must ensure that interface errors cannot propagate to affect the service of the component [LH94].

5.6 The Virtual Diagnostic Network

By exploiting the high-level virtual network service [OPK05b], a so-called *virtual diagnostic network* is established for the transport of diagnostic messages. Since only elementary interfaces [Kop99a] are used for the construction of virtual networks, no back-propagation of the diagnostic dissemination service to any safety-critical DAS is possible. An interface is called elementary, if the information flow and the interface control information are both unidirectional, i.e. the information producing subsystem can perform its information dissemination function without depending on any control signals from the information consuming subsystem. Otherwise the diagnostic subsystem would be elevated to a safety-critical subsystem that must be validated to a dependability of the highest criticality class.

5.6.1 Properties of the Virtual Diagnostic Network

In the following we discuss how the virtual solution for the dissemination of diagnostic information satisfies the requirements listed in Section 5.1.

Access to the Dissemination Service

In order to allow accurate diagnosis, all physical and functional entities of the integrated embedded system need to have access to the diagnostic dissemination service. This is guaranteed in the DECOS architecture, since all jobs (of all DASs) can access the virtual diagnostic network. Furthermore, all high-level services (e.g., hidden gateway service, fault-tolerance mechanisms) may disseminate diagnostic information via the virtual network whenever a symptom is detected.

Alternatively, one can compare the access to the virtual diagnostic network with a diagnostic gateway hosted at each component. Such a gateway implements a *diagnostic firewall* that restricts the type of information to be disseminated on the virtual network in such a way, that only data of diagnostic relevance (i.e. a symptom) is transmitted on this network.

Elimination of the Probe Effect

A purely virtual solution for the dissemination of diagnostic information has two main advantages. At first, real-time traffic is not compromised in any way, since the bandwidth for the exchange of diagnostic information is fixed a priori at design time. This way a deterministic message exchange for all non-diagnostic DASs is guaranteed. Secondly, the purely virtual solution ensures that no additional hardware faults are introduced due to wiring or connector problems. Consequently, no probe effect can be introduced [Gai86] and no additional errors can be introduced into the system.

Since, the virtual diagnostic network is like any other virtual network in the DECOS architecture an encapsulated communication service, a fault affecting this network cannot propagate to another virtual network [OPK05b, Obe04]. Even more, ensured by the construction of the virtual network service, no monopolization of the available bandwidth is possible. Thus, no jobs with a babbling idiot failure can negatively effect the dissemination of other messages within such an encapsulated network.

By contrast, consider the widely deployed Controller Area Network (CAN) protocol. Here, the dissemination of additional diagnostic messages may either cause the delay of real-time messages or, in case of low priority message, may exhibit substantial delay. Since the CAN standard [Bos91] does not provide a global time base, and no time stamping is available, such a delay renders detection of correlated failures difficult.

No Additional Hardware

Since the diagnostic dissemination service is a purely virtual solution, no additional hardware that may decrease system reliability due to connector and wiring faults is introduced. In addition, this virtual solution meets also the tight cost demands imposed by industry.

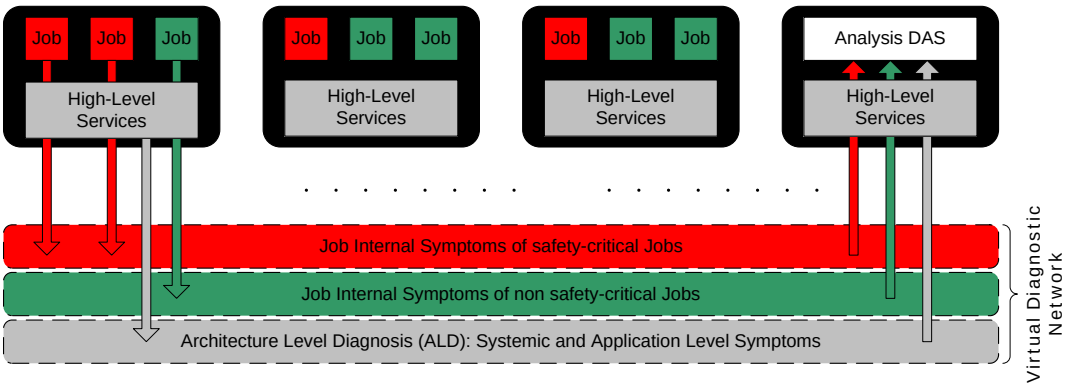

Figure 5.15: The Virtual Diagnostic Network

Referring to the previous CAN example, without such a virtual solution a possible alternative would be the deployment of a parallel diagnostic CAN network that will circumvent the highlighted problems, but will be highly unlikely due to economic constraints of the automotive industry. In addition, the additional hardware of a further network also increases the potential of wiring and connector faults that may decrease the accuracy of the analysis algorithms.

The idea of a second physical diagnostic network has been realized in the ISTORE project, where besides a primary data network, a dedicated diagnostic network for the dissemination of diagnostic information has been realized [BOK+99]. A physical CAN network is used for the dissemination of data from environmental monitoring sensors (e.g., temperature, fan RPM).

5.6.2 The Structure of the Virtual Diagnostic Network

The structure of the virtual diagnostic network is depicted in Figure 5.15. As illustrated, the virtual diagnostic network allows the dissemination of

- Job internal symptoms from jobs of the safety-critical subsystem (e.g., brake-by-wire DAS)
- Job internal symptoms from jobs of the non safety-critical subsystem (e.g., comfort DAS).
- Systemic and application-specific symptoms gathered by the architectural services (Architecture Level Diagnosis (ALD)) following the cross-checking principle

In accordance with [OPK05b], we deploy a virtual event-triggered communication service for diagnostic messages of non safety-critical jobs and the dissemination of architecture level diagnostic messages. Due to the typically event-triggered nature of diagnosis (we are only interested in state changes that indicate interface specification

Figure 5.16: The Diagnostic Message Format

violations) a communication service tailored to this type of communication is needed. The advantage of using an event-triggered virtual network is the better utilization of available resources, since diagnostic messages are typically only sporadically transmitted.

For the dissemination of job internal information of jobs from the safety-critical subsystem we utilize a virtual time-triggered communication service. This is necessary, since the DECOS architecture does not support event-triggered communication services for safety-critical subsystems by design [KOPS04]. This way, the SCU of a component needs not to provide any additional communication services for diagnosis. However, note that state information can be converted to event information in case an analysis algorithm operates on event semantics.

The third channel, reserved for ALD is the most important one. This channel transports the diagnostic messages constructed by the high-level diagnostic services following the cross-checking principle. One can see the diagnostic symptoms similar to event-triggered jobs that push diagnostic messages in case of an event of relevance to the virtual diagnostic network. At the receiver side, the analysis job(s) of the diagnostic DAS fetch the incoming messages from the respective input ports. Since this information is much more trustworthy than the job internal information (the interface state is revealed to all other components in the system and can thus be cross-checked; see also Section 5.5.2), more bandwidth is reserved for the ALD than for the other two information sources.

5.6.3 The Diagnostic Message Format

As described, the available information at each component for the purpose of diagnosis needs to be transferred to the analysis subsystem in order to allow judgment about the nature of possible faults affecting the system. In order to allow this exchange of information a diagnostic message format needs to be defined that includes information, about the time, space, and value domain in accordance with Section 5.5. Since the analysis process operates on the global system state it must be possible to encode not only value and time information into the message, but also include the space dimension (e.g., for the discrimination between hardware and software faults). Furthermore, in order to assess the health status of nodes the inclusion of data values into the diagnostic frame for subsequent analysis must be possible (e.g., a sensor value, the clock state correction term). In addition, the diagnostic frame format as depicted in Figure 5.16 is not limited to one specific type that may not be applicable in all domains. Consequently, different types can be specified using the message type field to allow a flexible design of the messages (i.e. extensibility). Furthermore,

Field Name	Semantics
Message Type	The message type field contains the type of the diagnostic message. It allows a flexible definition of diagnostic messages.
Symptom ID	Each symptom is identified by an unique identification.
Time Domain	This field includes a timestamp that allows tracing of correlated failures/anomalies.
Space Domain	This field encodes the space information, i.e. cluster, component, subsystem, DAS, and job information. The space information allows including physical and functional information into the analysis process.
Value Domain	The value domain field allows the inclusion of particular values of interface state variables at the moment of the execution of the detection mechanism into the analysis process.

Table 5.1: Diagnostic Message Format

bandwidth is in general seen as a scarce resource. As a consequence, the efficient use of available bandwidth is of importance. Depending on the type of diagnostic message (in combination with a particular symptom ID), not all data fields (time, space, value) need to be transferred. For example, in case of a boolean check, no value field needs to be transmitted. The design of the diagnostic message with detailed explanation of each message field is depicted in Table 5.1.

5.7 Specification and Execution of Out-of-Norm Assertions

Since ONAs encode fault patterns on the distributed state of the system, methods that allow capturing of the value, time, and space domain are needed. For this reason we use timed automata for both the specification of symptoms (i.e. component/job local checks on the interface state) and the consecutive analysis process.

A timed automaton [AD94], i.e. a state transition graph annotated with timing constraints, has an intuitive syntax and semantic that makes it especially interesting in the specification and design of diagnostic algorithms. Furthermore, using timed-automata as the specification method has also the significant advantage of having a representation that can easily be transformed into a machine executable form.

As depicted in Figure 5.11 an ONA consist typically of a number of symptoms defined on the interface state of a component (or port state of a job). This information is then combined over time in order to analyze the type of fault affecting the given system. In the following we specify a timed symptom/analysis automaton formally and present examples illustrating this concept on the basis of the DECOS architecture.

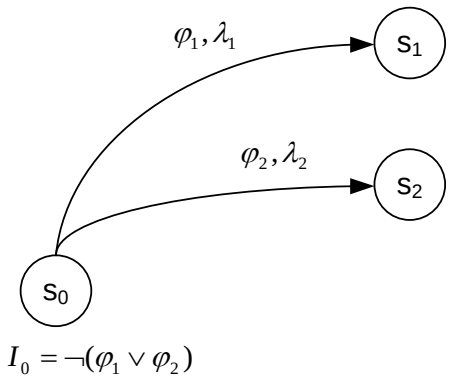

Figure 5.17: A Timed Symptom/Analysis Automaton

5.7.1 The Timed Symptom/Analysis Automaton

Since the timed automata used for the specification of ONAs need to be executable, we restrict timed automata defined by [AD94] to be nonzeno (i.e. there is no infinite sequence of transitions without any progression of time in between), deterministic and require that the invariant of each location is complementary to the guards of all outgoing edges, i.e. an edge must be taken as soon as possible. In the following we extend the definition by [AD94] to be suitable for our purposes.

Guards and Actions. In order to formally define the timed symptom/analysis automaton, we specify what constraints are allowed as the enabling conditions called *guards* of a timed automaton. We define the set $\Phi(X, V, M) : (X, V, M) \mapsto (\mathbb{T}, \mathbb{F})$ of guards for a set of clocks X, variables V, and messages M via the following grammar:

$$\varphi := x \circ c \mid x \circ v \mid x \circ m \mid v \circ c \mid v \circ m \mid v \circ v' \mid m \circ m' \mid \text{av}(m) \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2,$$

where x is a clock in X, c is a constant in $\mathbb{N}$, v and v' are variables in V, m and m' are messages in M. $\circ$ is a binary relation $(\leq, <, =)$. $\text{av}(m)$ tests whether a message m is available at the respective input port (only needed for event-triggered communication).

The set of *actions* $\Lambda(X, V, M) : (X, V, M) \mapsto (X, V, M)$ is defined via the following grammar:

$$\begin{aligned}\lambda := \\ & x := c \mid x := m \mid v := c \mid v := m \mid v := v' \mid m := c \mid m := v \mid m := m' \mid \\ & m_{\text{diag}}! \mid m? \mid \lambda_1; \lambda_2,\end{aligned}$$

where x is a clock in X, c is a constant in $\mathbb{N}$, v is a variable in V, and m is a message in M. $m_{\text{diag}}! \in M$ is a diagnostic message to be disseminated via the virtual diagnostic network, and $m?$ reads message m from the respective input port. Note, that in case of event-triggered communication $m?$ is consuming, in contrast to $\text{av}(m)$.

Definition of the Syntax. A timed symptom/analysis automaton A is a tuple $\langle L, L^0, X, V, M, I, E \rangle$, where

- L is a finite set of locations,
- $L^0 \in L$ is the initial location,
- X is a finite set of clocks,
- V is a finite set of variables,
- M is a finite set of messages,
- I is a mapping that assigns to each location an invariant as a constraint in $\Phi(X, V, M)$ $(I : L \mapsto \Phi(X, V, M))$,
- $E \subseteq L \times \Phi(X, V, M) \times \Lambda(X, V, M) \times L$ is a set of transitions. A transition $\langle s_1, \varphi, \lambda, s_2 \rangle$ represents an edge from location s_1 to location s_2 with guard φ. The guard is a constraint of clocks X, variables V and messages M and determines when the transition is enabled. The action λ is a set of assignment and message operations to be performed. Figure 5.17 depicts an example automaton.

5.7.2 Symptom Specification

As defined in Section 5.5, symptoms are the local manifestation of fault patterns at a single component defined via ONAs. In the integrated architecture we distinguish between symptoms that are specified in the context of the functional and physical structure. While the first ones are application-specific and generated by monitoring all incoming and outgoing messages at the ports of jobs, the latter are system-specific and generated by the core and higher-level services for the detection of hardware failures.

- **Application-specific Symptoms.** An application-specific symptom detector monitors the behavior of the job at one or more ports of a link of the respective virtual network [OPK05b]. Whenever it detects a violation of the link specification, the symptom detector sends a diagnostic message to the analysis subsystem via the virtual diagnostic network.
- **Systemic Symptoms.** Systemic symptom detectors monitor the interface state of the component instead of the port state of the jobs hosted on the component. These hardware checks include for example the state of the membership, the clock correction term, or the health state of the connection to the physical network (e.g., message reception on all replicated physical channels).

By correlating the information from both, application-specific and systemic symptoms a finer differentiation between hardware and software faults is possible, thus overcoming today's ECU centered diagnosis schemes.

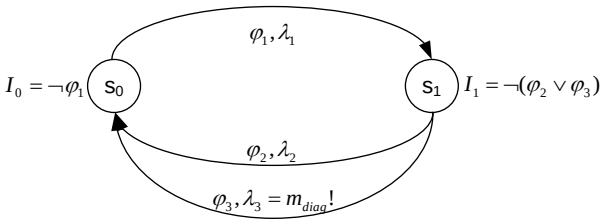

Figure 5.18: Systemic Symptom Detection

Systemic Symptom Specification

Typically, systemic symptoms are checks on interface state variables of components in order to determine correctness of the underlying physical time-triggered communication system or component hardware. Consequently, the guards of systemic symptoms are defined on the CNI of the deployed communication controller mapping relevant interface state variables to be externally readable. Once defined systemic symptoms that proofed to be a good choice in the field can be reused unmodified in future systems that deploy the same platform (i.e. core network architecture). This way expensive and time consuming revalidation of systemic symptom detection is not necessary. Since the major computational overhead needed for analysis is shifted to the diagnostic DAS, the node local check can be kept simple and the overhead reduced to a minimum.

Systemic symptoms heavily exploit the slot-based nature of time-triggered communication systems for the performed checks. The regularity (i.e. the a priori knowledge of the message sending instants) of the communication systems simplifies the construction of symptoms, since actions in the timed automaton are triggered by the progression of real-time. Thus, systemic symptom detection is executed either in any slot, or once during the TDMA round depending on the type of check to be performed.

A schematic symptom detector is depicted in Figure 5.18. Since the progression of the automaton is triggered by the progression of real-time the guard φ_1 restricts the points when the transition from location s_0 to s_1 can be taken to slot granularity. The invariants I_0 and I_1 force to take a transition to be taken whenever a guard is enabled. The guards φ_2 and φ_3 are encoding checks on the interface state variables of the component to detect deviations from the interface specification such as the reception of messages on all replicated channels. Typically, the guards φ_2 and φ_3 are complementary, i.e. $\varphi_2 = \neg\varphi_3$. In case guard φ_3 is enabled, a corresponding diagnostic message $m_{\text{diag}}!$ is constructed and disseminated via the virtual diagnostic network (i.e. $\lambda_3 = m_{\text{diag}}!$).

For some real-world examples for systemic symptoms on the basis of the DECOS system refer to Section 6.3.

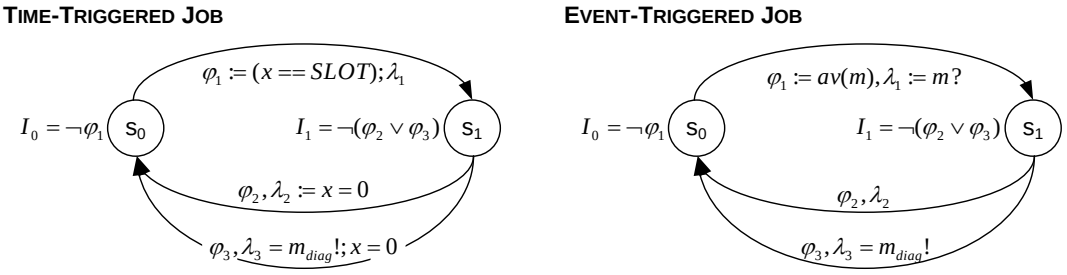

Figure 5.19: Application-Specific Symptom Detection

Application-Specific Symptom Specification

While systemic symptoms are checks on interface state variables of a component, application-specific symptoms are evaluated against the port state of the jobs hosted on a component. Thus, the guards are defined of the content of messages. In contrast to systemic symptoms, application-specific symptoms highly depend on the application context and need thus be devised by the application developers. Only the deep understanding on the dynamics of the application and relations between the different jobs allows defining meaningful symptoms.

One has to differentiate between time-triggered and event-triggered application-specific symptoms as depicted in Figure 5.19. In case of time-triggered symptoms the guard φ_1 enabling the evaluation of the port state variables are solely triggered by the progression of time $\varphi_1 := x = \text{SLOT}$, where $x \in X$ and SLOT denotes the length of a TDMA slot of the corresponding cluster schedule. By contrast, in case of event-triggered jobs, the enabling condition, i.e. the guard φ_1 depends on the availability of a message $\varphi_1 := av(m)$. The action λ_1 then reads the respective incoming/outgoing message $\lambda_1 := m?$ for further analysis encoded in the outgoing transitions from location s_1. In analogy to the systemic symptom detector, once a violation of the port specification is detected by enabling guard φ_3 a corresponding diagnostic message $m_{\text{diag}}!$ is generated and disseminated. In case φ_2 evaluates to $\mathbb{T}$ no violation has been detected and the execution of the automaton starts over again.

Automotive Example

Today many cars are equipped with cruise control functionality. Cruise control enables the automatic sustaining of the car's speed despite variations in uphill and downhill grade of the road. Adaptive Cruise Control (ACC) is further improvement of this automotive functionality allowing reaction to changing traffic situations in order to improve driving comfort and to reduce the number of crashes [Jon01, LS04].

The ACC ECU processes radar information in order to determine whether the clearance between the vehicle and the forward vehicle is within the limits. If the threshold is violated then control signals to the engine control ECU as well as the brake control ECU are sent. As a consequence the brakes of the car are applied and the vehicle is slowed down (with a maximal brake force of 25% of the available brake

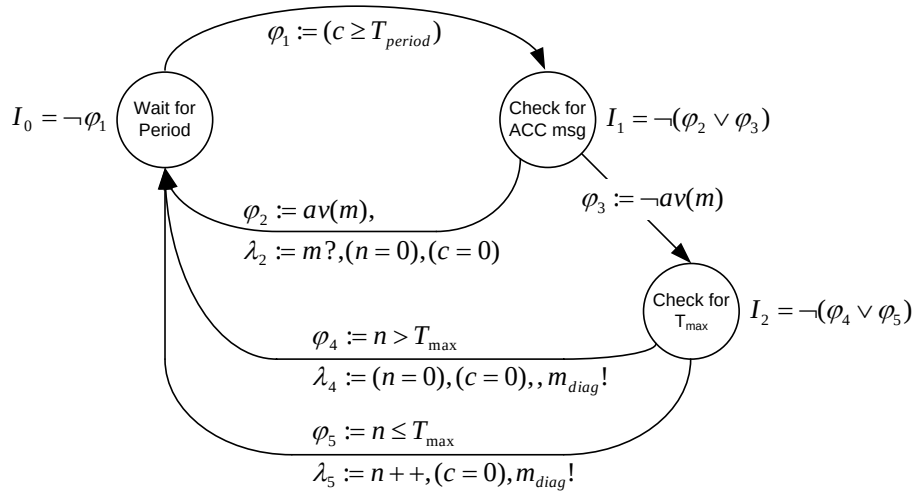

Figure 5.20: Adaptive Cruise Control - Error Detection at the Receiver

power). Whenever the radar indicates that the forward vehicle is no longer in the vehicle's path, the vehicle resumes to the driver's intended cruising speed. This way the ACC allows reaction to changing traffic environments by adapting the vehicle's speed accordingly. According to [Jon01] an ACC systems consists of:

- The ACC or Headway ECU: The functionality of this ECU is to determine control signals for the engine control ECU and brake control ECU on the information provided by the radar module.
- Engine Control ECU: This ECU processes the information from the ACC ECU and adapts the engine's throttle accordingly.
- Brake Control ECU: This ECU decelerates the vehicle whenever a request from the ACC ECU is received.
- Instrument Cluster: This subsystem informs the driver about the state of the ACC system and forwards input from the driver to the ACC ECU and engine control ECU.

Today's ACC systems employ a CAN communication infrastructure, although future systems will most likely be implemented on top of a time-triggered architecture [LS04]. In order to improve system reliability a periodic communication schedule is implemented. In this configuration, each node is required to send a message periodically (e.g., every 30 ms) as a life-sign and to disseminate updated sensory information. In case a timing violation of the interface specification repeatedly occurs, the ACC service is shut down to preclude any hazardous situation due to possible system malfunctions. In contrast to a time-triggered communication system, a loss of a message does not imply a failing sender, but can also result from bus inaccessibility times due to the dynamical resolving of contention. Consequently, a shutdown of a service typically requires more than one missing message at the receiver.

Figure 5.20 illustrates a symptom detector that checks for this timing violation at the brake control ECU. Every period T_{period} the timed automaton checks whether a message from the ACC ECU has been sent. In case no message is available ($\neg av(m)$) at the respective input queue, the timed automaton checks whether a violation of the maximum number of consecutive messages failures specified in $T_{\max}$ has occurred. If n, storing the number of consecutive missing messages, is smaller or equal to $T_{\max}$ a diagnostic message is sent to indicate this out-of-norm behavior. By contrast, in case $n > T_{\max}$, a diagnostic message is sent indicating a system failure.

5.7.3 Analysis Specification

In analogy to the specification of symptoms, for the encoding of analysis algorithms as introduced in Section 3.7 also timed automata are used. This flexible way of defining the analysis algorithms allows combining different analysis techniques to improve the overall performance of the analysis process.

A simple α-count scheme is depicted in Figure 5.21. The transition from location s_0 is taken whenever a diagnostic message of relevance for this ONA is available at the input port, formally $\varphi_1 = av(m_{\text{diag}})$. Once a message is available at the port, the action λ_1 initializes the variables for the analysis process (e.g., resetting the α-counter) and sets clock ($x_i \in X$) variables to the respective initialization values. This is an important step in order to include timing information, i.e. the length of the interval between two consecutive message reception indicating the same symptom, into the analysis process. For taking an edge from state s_1, one of the following guards need to be enabled:

- φ_1: in case the α-counter value is below the defined threshold T_α and the time between two consecutive message receptions is also smaller than an a priori defined threshold value Γ_α a transition to state s_2 is made as soon as a further diagnostic message is available $av(m_{\text{diag}})$.

- φ_2: in case the α-counter value is below the defined threshold T_α but the threshold value Γ_α for two consecutive message receptions is exceeded, the transition to state s_3 is taken.

- φ_3: in case the α-counter value is $\alpha = 0$, the transition to state s_5 is taken and the fault pattern discarded. After updating the statistics of the analysis process λ_8, the automaton starts over again by progressing to state s_0.

- φ_4: in case the α-counter value is greater or equals the threshold T_α, a fault pattern has been detected and the transition to state s_4 is taken. After updating the statistics of the analysis process λ_7, the automaton starts over again by progressing to state s_0.

The most important difference between the diagnostic strategy of the DECOS integrated architecture and the prevalent diagnosis scheme found in the majority of

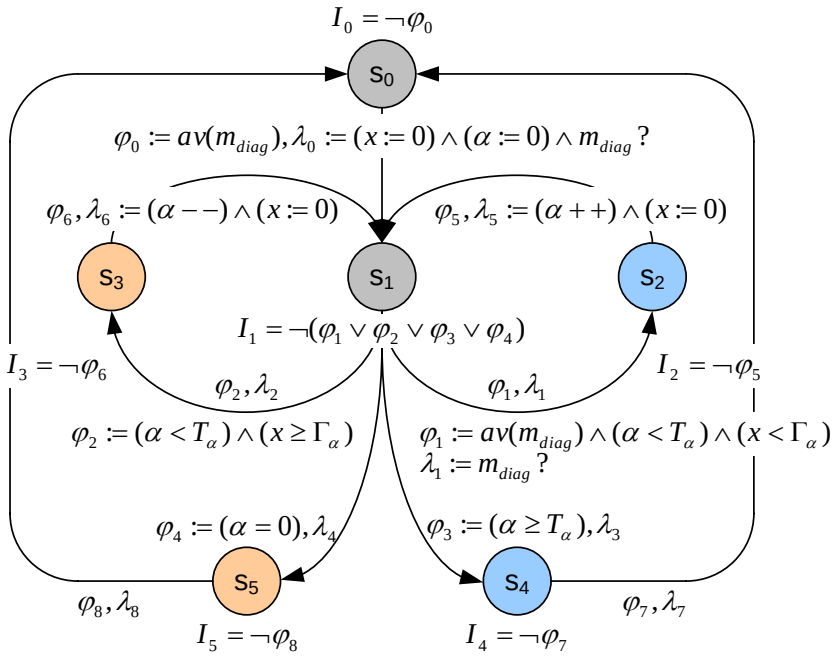

Figure 5.21: Timed Analysis Automaton

deployed systems is the fact, that in DECOS correlated information is rigorously exploited to improve the accuracy of the analysis process and consequently put an end to unnecessary replacement actions at the service station. The availability of the global time base simplifies the identification and analysis of correlated effects on the distributed state significantly.

As depicted in Figure 5.22 the timed analysis automaton specified in Figure 5.21 can be refined to allow including information about correlated symptoms into the analysis process by exploiting the availability of a global time base. The timestamps of the messages (i.e. the instant in time the symptom has been detected) is used to check whether other diagnostic messages indicate the same problem. In order to compensate for small variances, we do not check for a particular instant in time, but for a small interval of time $[-\frac{\Delta}{2}, +\frac{\Delta}{2}]$, where Δ can be set individually (e.g., slotlength, TDMA round length). Instead of increasing the α-counter just by one, a weighted increment allows the inclusion of correlated information in the analysis process.

As shown in Figure 5.22 location s_1 has four outgoing edges in analogy to the automaton specified in Figure 5.21. Once a diagnostic message m_{diag} is available $av(m_{\text{diag}})$ and both the α-counter threshold and timing constraint are not violated the automaton progresses to location s_2. Location s_2 has three outgoing edges for the following possibilities:

- **New message.** In case the timestamp of the diagnostic message is greater than the correlation timestamp $m_{\text{diag}} > t_{\text{corr}} + \frac{\Delta}{2}$, the counter value *new* is incremented and the timestamp for possible correlated messages updated according to to the timestamp of the received message $t_{\text{corr}} := m_{\text{diag}}.\text{timestamp}$.

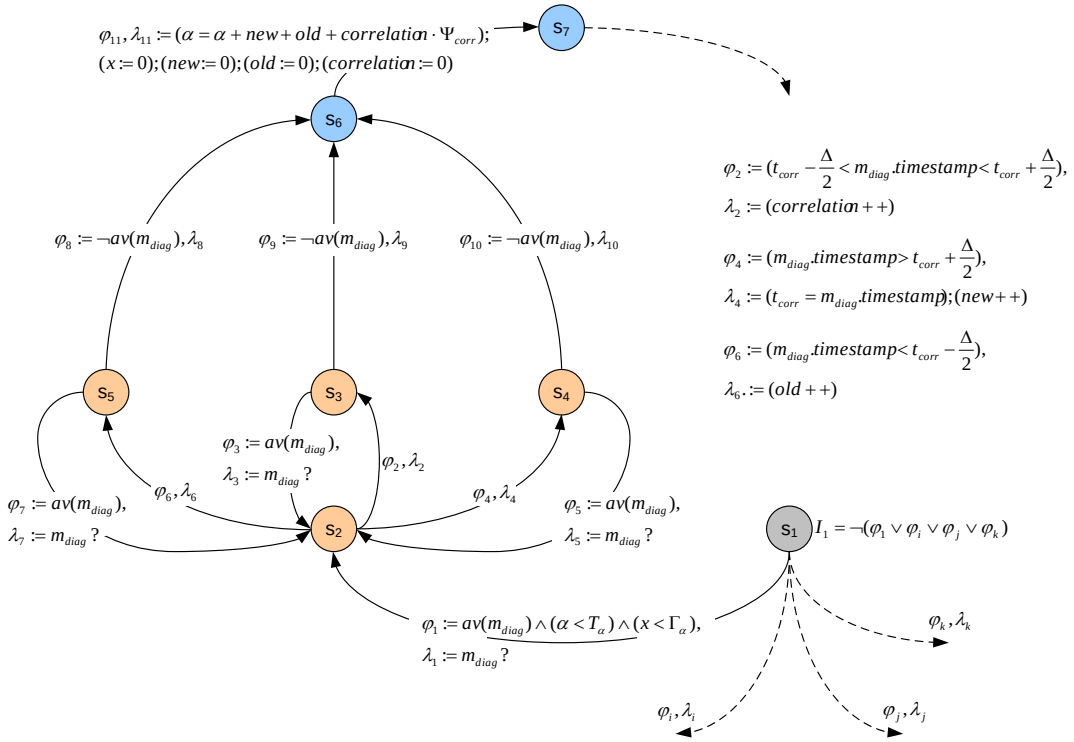

Figure 5.22: Correlation of Information

- **Correlated message.** If the timestamp of the received diagnostic message m_{diag}.timestamp is within the interval $[t_{\text{corr}} - \frac{\Delta}{2}, t_{\text{corr}} + \frac{\Delta}{2}]$, guard φ_2 evaluates to true the transition to location s_3 is taken. The action λ_2 increments the counter value *correlation*.
- **Old message.** In case φ_6 is enabled, an message with an timestamp $m_{\text{diag}} < t_{\text{corr}} - \frac{\Delta}{2}$ is received. This message provides diagnostic information that is already outdated. The *old* counter is updated accordingly and the automaton progresses either to location s_6 or s_2 depending on the availability of other messages. Note, that delivery of outdated information can be a result of the use of an event-triggered virtual network service (e.g., peak load scenario).

In case no more messages are available $\neg av(m_{\text{diag}})$, the α-counter value is incremented and all associated variables set to their initial values. The updated α-counter value is set to $\alpha = \alpha + \text{new} + \text{old} + \text{correlation} \cdot \Psi_{\text{corr}}$, where Ψ_{corr} is the a priori defined weighting factor (the higher the value of Ψ_{corr}, the more impact correlated detections have on the α-counter). By using weighted increase values, correlation in different domains can be taken into account. For example, information about nodes having physical proximity can be used for analysis of external faults (e.g., correlated node faults in case of heavy EMI).

For exemplary analysis automata as part of ONA definition on the basis of the DECOS system refer to Section 6.4.

```
clocks            X
variables         V
messages          M
current location  s

T = T_activation − n
while ( T < T_activation )
  while (∃ ⟨s_1, φ, λ, s_2⟩ ∈ E with φ = 𝕋 ∧ s = s_1)
    (X, V, M) := Λ(X, V, M)   //execute action
    s := s_2                  //new location
  end
  T = T + 1                   //advance time
  ∀x ∈ X : x = x + 1
end
```

Figure 5.23: Execution Step (n ticks) of the Timed Automaton A

5.7.4 Execution of the Timed Automata

The timed symptom detection/analyisis automata are executed on the action lattice of the sparse time base. In the silence interval with respect to the communication services both the timed automata for symptom detection and analysis as part of an ONA are executed. See also Figure 5.24 for a graphical representation of the execution scheme. The algorithm is presented in Figure 5.23 and works as follows:

1. The time variable is set to the beginning of the last interval of activity of the sparse time base with respect to the action lattice for virtual network service $T = T_{activation} - n$, where $T_{activation}$, is the actual global time. n denotes the number of ticks elapsed during on interval of silence and activity.

2. As long as the simulation time T is smaller than the actual time $T_{activation}$, the execution continues. In case this condition does not hold, the execution of this automaton is terminated for this activation cycle.

3. A transition is taken whenever a guard φ is enabled. In case no guard evaluates to $\mathbb{T}$, the following step (4) is skipped.

4. If a transition is taken, all corresponding actions λ are executed (e.g., updating of variables, message reception) and the current location is updated.

5. Both the global simulation time T and local clock variables X are increased by 1. Continue with step 2.

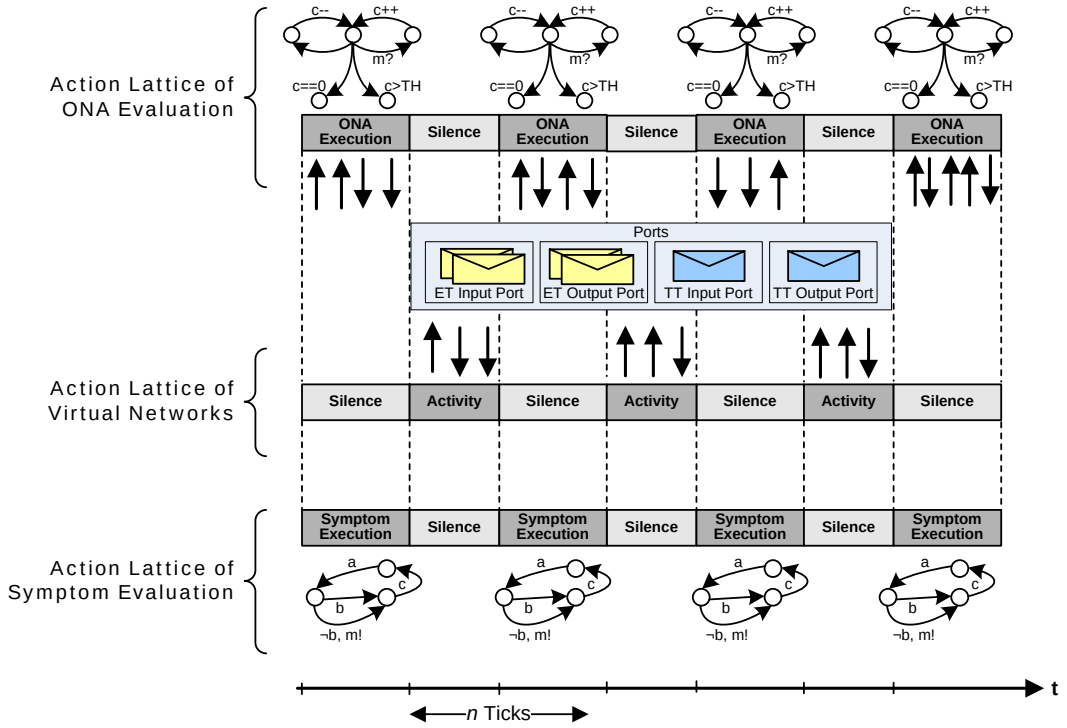

Figure 5.24: Timed Automata Execution on the Sparse Time Base

5.8 Detection at Component Level

The connector unit implements the high-level services of the DECOS architecture as introduced in Section 4.3. In accordance with the high-level services a connector unit can be subdivided into a set of layers, each implementing a particular service (see Figure 5.25). The *allocation and virtual network layers* establish the subdivision of network resources, the *gateway layer* allows the controlled import and export of information between DASs, the *fault-tolerance layer* implements voting strategies in order to tolerate failures in the value domain, and the *message classification layer* monitors all incoming and outgoing messages of jobs. Finally, the *API layer* as part of the application middleware allows dissemination of job internal diagnostic information that can be included into the analysis process to improve the accuracy of the assessment algorithms. In the following we discuss the layers of a connector unit in detail, thereby focussing on symptom detection at each layer. We describe the type of diagnostic information that can collected at each layer and can help in the identification of faults affecting the system. Furthermore, we discuss how the automata for symptom detection are executed and the detailed model of the virtual diagnostic network at component level.

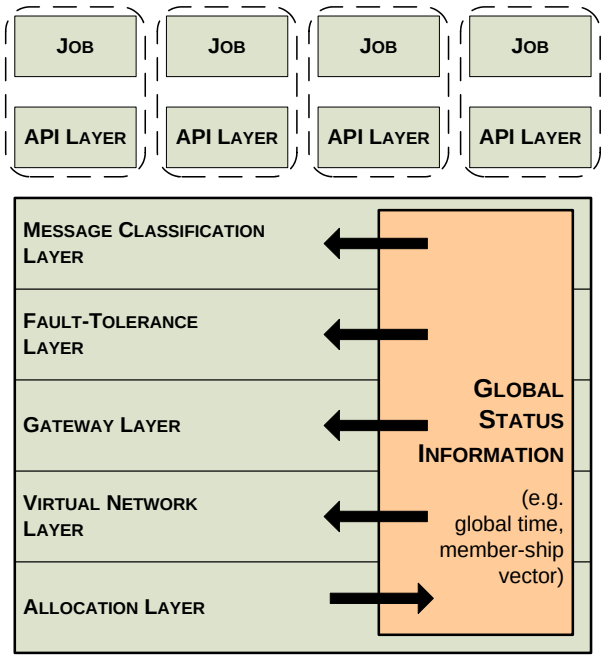

Figure 5.25: Layers of a Connector Unit of a DECOS Component

5.8.1 The Allocation and Virtual Network Layer

The BCU splits the available bandwidth of the underlying time-triggered core network between the safety-critical and non safety-critical subsystem of a component. Furthermore, status information of relevance is updated by the allocation layer (e.g., global time, membership vector). The allocation layer now recursively splits this remaining bandwidth between the virtual networks of the subsystem a particular component has access to. While the underlying allocation layer is shared between all jobs of a subsystem hosted on the component, regardless of the DAS these jobs belong to, the virtual network layer and all higher layers comprise a dedicated instance of architectural services for each DAS with jobs hosted on the component.

The task of the virtual network layer is the establishment of a virtual communication infrastructure according to the control paradigm of the respective DAS [OPK05b]. Depending on the employed paradigm (event-triggered vs. time-triggered) the interface to the upper layers are either queues for messages with event semantics or a temporal firewall interface [KN97] for messages with state semantics. Queues support exactly-once processing of messages while memory elements storing state messages are overwritten whenever a more recent version arrives. The dimensioning of the queues depends on the parameters derived from a communication model that implements the assumptions about interarrival and service times. Since the temporal specification is probabilistic (e.g., message consumption/production rate [Kle75]) and typically described by asymptotic probability distributions, a classification of message according to a bivalent logic is hardly achievable.

However, in case these assumptions do not hold in reality, queue overflows will occur, that may cause jobs to fail. This non conformance with the temporal specification represents valuable information for the developers of the system. Sporadic system failures resulting from misleading assumptions about the rates of message in an event-triggered network fall into the class of Heisenbugs [Gra86]. Thus, queue overflows are important engineering feedback for the dimensioning of the deployed queues and fall into the class of job borderline faults according to the maintenance-oriented fault model introduced in Section 5.3. As a consequence, systemic symptom detectors at the virtual network layer are defined on the interface state variables of the virtual network service.

From a diagnostic point of view time-triggered communication is much better diagnosable due to its deterministic nature and a priori defined communication schedule. By contrast, the flexibility of event-triggered communication comes with the price that symptoms are defined on a probabilistic communication model. By exploiting the underlying time-triggered core network, the error detection capabilities for event-triggered communication can be increased (i.e. the availability of a global time base allows monitoring of adherence to the temporal specification). However, for circumventing the result of [FLP85] not only the communication system must be monitored, but also the execution environment to judge whether a particular event-triggered jobs has also failed. Since the DECOS architecture decouples the communication system from the execution system of the jobs, such a diagnosis is possible.

5.8.2 Gateway Layer

The gateway layer realizes the architectural gateway services and performs the redirection of messages between virtual networks with the necessary property transformations. The controlled export and import of information between the DASs allows tactic coordination and an optimal usage of available resources (e.g., sensor data) [OPK05a].

The gateway layer also supports forwarding of diagnostic messages of physical clusters attached to the DECOS system. For instance, if a LIN fieldbus is connected to a DECOS component, the diagnostic information about the health status of this external physical fieldbus can be included into the analysis process. However, note that such a physical gateway needs to provide error containment capabilities in order to retain the encapsulation properties of the virtual diagnostic network.

Central to the gateway is the *gateway repository*, i.e. a real-time database, that decouples the two virtual networks. In [OP05] the specification and execution of virtual gateways is described in detail. Like for diagnosis a specification method as an extension to deterministic timed automata is proposed. This similarity is due to the fact, that one can see the diagnostic services hosted on the components as a gateway service to the diagnostic DAS.

5.8.3 Fault-Tolerance Layer

The task of the fault tolerance layer is mainly voting on replicated messages from different jobs hosted on physically separated components. In case of diverging replicated messages, a corresponding diagnostic message is constructed and forwarded to the analysis DAS indicating the violation of the port state specification. In case of X-by-wire systems the condition monitoring of the fault-tolerance mechanisms is of great significance and required by maintenance guidelines like ARINC 624 [Aer93].

Monitoring of fault-tolerance mechanisms cannot be done at job level in case of transparent fault-tolerance techniques [Bau01], since a faulty interface state is not forwarded to the jobs but discarded at the fault-tolerance layer. Consequently, this information must be captured by systemic diagnostic checks at lower layers.

For example consider a fault-tolerant steer-by-wire system [Hei03]. Such a system typically consists of three redundant actuators, mechanically voting on the input of the redundant ECUs and thereby controlling the axle of the front wheels. The redundant ECUs compute the actual steering angle based on the commands of the driver and the car dynamics subsystem of the car. In case of a component failure the system does not provide a degraded service mode. However, the driver needs to be informed by a corresponding MIL activation and the defective component needs to be replaced at the service station as soon as possible.

By continuously monitoring the state of replicated units condition-based maintenance strategies can be realized. In case a component shows indication for premature wear-out (e.g., increase of the transient failure rate) the service technician can change the component preventively, thus not only keep the safety level of the system to the optimum but also maintain the driver's trust into the computer system of the car [Ber02].

5.8.4 Message Classification Layer

The task of the message classification layer is to check whether an incoming or outgoing message of a job conforms to its interface specification. Consequently, the message classification layer monitors all incoming and outgoing messages in order to reveal any out-of-norm behavior specified by the symptoms of ONAs indicating potential software or hardware faults (e.g., sensor faults).

In case such an untimely or value-incorrect message has been detected [GIJ+02], a corresponding diagnostic message is forwarded by the virtual diagnostic network to the diagnostic DAS for subsequent analysis. Note, that the message classification layer does not modify or delete any message. This is in the scope of the application jobs only, since otherwise diagnosis would be elevated to the highest criticality class.

5.8.5 Application Programming Interface (API) Layer

The API layer as part of the application middleware establishes the data structures and the function calls via which the application accesses the architectural services.

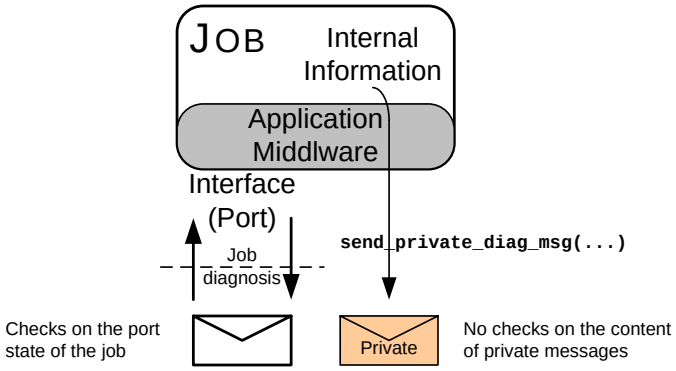

Figure 5.26: API Layer

For example, the virtual network layer offers a generic event-triggered communication service. In case a job expects a CAN [Bos91] specific interface, the generic service is accessed through a middleware that maps the generic interface onto a CAN specific one that can be accessed by native CAN function calls.

In order to allow the dissemination of job internal information that has been decided not to make available at the interface state for complexity or intellectual property considerations, the diagnostic dissemination service is required to provide a dedicated channel that allows the transferring of such information to the analysis DAS. The deployed API middleware provides access to this virtual diagnostic network for the dissemination of job internal information by providing a dedicated function call (e.g., `send_private_diag_msg(t_diagmsg internal_info)`).

As depicted in Figure 5.26, a job can utilize the diagnostic function call of the API provided by the application middleware in order to send job internal diagnostic information. Job internal information can be important when discriminating between job inherent faults according to the maintenance-oriented fault model as introduced in Section 5.3. For example, in case a sensor provides self-test functionality, this information can be read by the job and forwarded to the diagnostic subsystem to indicate potential sensor failures.

5.9 Analysis – Determining the Maintenance Action

In the following we discuss the structure of the deployed diagnostic DAS where the analysis algorithms are executed in order to determine the correct replacement strategy. Furthermore, we discuss how this strategy allows solving of prevalent diagnostic problems.

5.9.1 Diagnostic DAS

The analysis of the gathered diagnostic information (i.e. symptoms) is shifted into a designated DAS - the diagnostic DAS - to ensure that the diagnostic subsystem

cannot interfere with jobs of other DASs and to not restrict the choice of implementations (e.g., central diagnostic component vs. distributed solution). In the following we briefly discuss the structure of the diagnostic DAS and a key strategy for the implementation of analysis algorithms to allow both a centralized and distributed implementation.

Structure of the Diagnostic DAS

In analogy to the DAS concept introduced in Section 4.1, the diagnostic DAS comprises one or more jobs. In case a distributed solution is favored the jobs are interconnected by a virtual network to allow dissemination of results and subsequent voting on the derived analysis. A distributed solution has the advantage of being more robust with respect to failures affecting the components hosting the jobs of the diagnostic DAS at the cost of increased complexity due to the need of agreement between the replicas. This system structure is depicted in Figure 5.27. A gateway

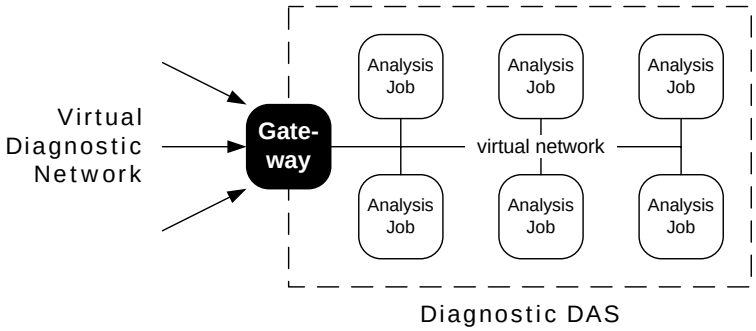

Figure 5.27: The Diagnostic DAS

forwards the information of the virtual diagnostic network to the job(s) of the diagnostic DAS. By exploiting the high-level services for the realization of the diagnostic DAS, a certification for ultra-dependability is not needed, since it can be guaranteed that there are no interdependencies between the diagnosis subsystem and the other applications.

Ground State

By executing the analysis algorithms as part of the ONA specification (see Section 5.7) in order to identify a possible fault pattern affecting the system, one has to consider that the result of the analysis needs to be stored permanently in non volatile memory to:

- Allow continuous analysis for more than one cycle of operation
- Be robust in case of a transient failure affecting the node hosting one or more jobs of the diagnostic DAS.

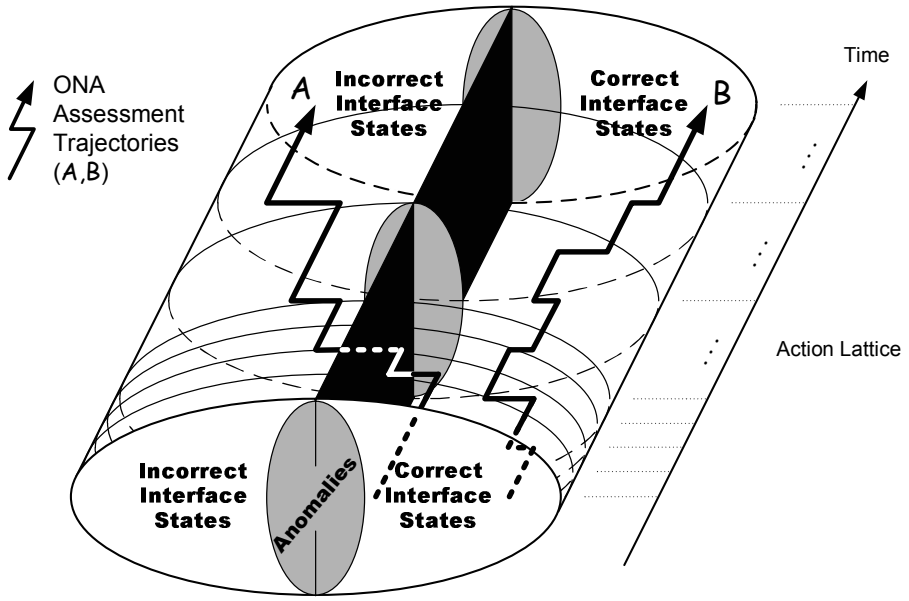

Figure 5.28: LRU Assessment Process

Consequently, the jobs need to be designed to periodically reach a ground state (e.g., every 1000 TDMA rounds) where a temporary result of the analysis computations is stored in non volatile memory. After a restart, either for a new cycle of operation or in case of a transient failure, this information serves as the initialization state of the job [Kop97]. Based on this data structure storing the latest judgment of the diagnostic DAS on the health state of the system, the new h-state is dynamically updated and made permanent once again the ground state is reached. However, one has to be aware, that at least to separated memory regions need to be available, since a failure might prevent a job to finish updating the information stored in the non volatile memory. This strategy has three significant advantages:

- **Robustness against transient failures.** In case a a transient hits the component hosting diagnostic jobs only the update of the health state of the system from the last ground state to the actual point in time is lost.

- **Facilitating a distributed solution.** Since at ground state a reduction of information is established, in case of a distributed solution, this information exchange between the jobs of the analysis DAS only requires a small fraction of the available bandwidth.

- **Optimal use of resources.** In present day automotive systems non volatile memory is a scarce resource. With the introduced strategy, the amount of information that needs to be stored can be limited.

5.9.2 Determining the Replacement Strategy

The fault model introduced in Section 5.3 serves as the basis for the assessment process. The diagnostic subsystem executes algorithms on the gathered diagnostic information in order to assess the condition of each FRU. The evaluation process performed by the diagnostic DAS is illustrated in Figure 5.28. The evaluation process is based on a consistent notion of state, which is provided through the *action lattice* of the sparse time base established by the core services of the integrated architecture. The arrows in Figure 5.28 indicate the LRU assessment trajectories. At first both arrows show conformance with the LRU specification. As time progresses arrow A exhibits an increasing confidence for a violation of the specification, while arrow B indicates a LRU behavior in accordance with the specified service.

The introduced integrated architecture provides a finer granularity of diagnostic information than federated systems. The assessment process exploits this knowledge about the functional and physical structure of the integrated architecture. The decomposition of the overall system into DASs with respective jobs is a key element for a more precise differentiation of experienced faults. By including the three dimensions of time, value, and space into the judgment process, a discrimination into the fault classes identified by the maintenance-oriented fault model is possible.

As a result of the diagnostic judgement process, a *trust level* for each FRU of the system is determined that forms the basis for decision-making process of the maintenance engineer. Figure 5.29 summarizes the maintenance actions for each fault class: As stated, we distinguish between component internal, borderline and external faults. Component internal faults are further categorized into job inherent, job borderline, and job external faults. The maintenance actions for each fault class that is discriminated by the diagnostic services of the DECOS architecture are summarized in the following

- **Component External.** In the proposed model we consider the persistence of external faults as transient. Consequently, in case of component external faults no maintenance action has be to taken.

- **Component Borderline.** Borderline faults require a closer inspection by the service technicians. Connector problems, are difficult to trace, since the inspection itself can be the corrective action [KHTP99]. In case of connectors showing wearout phenomena such as fretting or corrosion, a replacement will be necessary.

- **Component Internal/Job External** Component internal faults such as a crack in the PCB or a defective processor require the replacement of the component (i.e. the ECU in the automotive domain or a LRM in avionics).

- **Job Borderline.** Job borderline faults require the update of the configuration data of the virtual network service of the DAS.

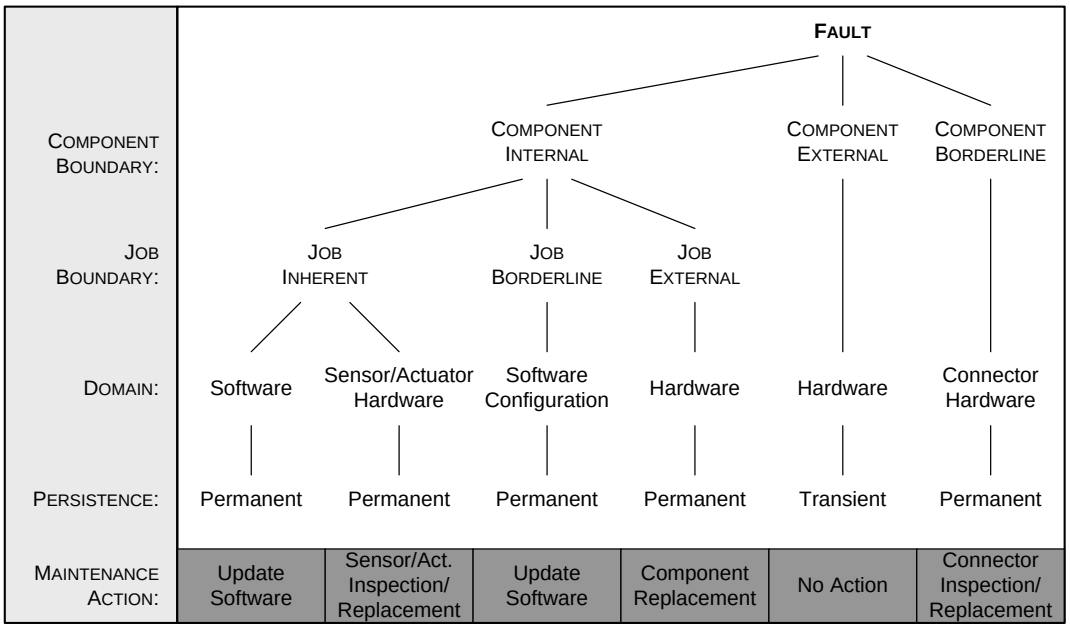

Figure 5.29: Determining the Maintenance Action for each Fault Class

- **Job Inherent.** Sensor/actuator faults require further inspection by the service technician in order to decide whether part replacement due to wear-out (e.g., change of brake pads) or a replacement of the transducer is necessary.

  Software faults identified by the diagnostic system requires an update of the job software in case this identification has also been acknowledged by the OEM and a corrected version of the job has been distributed to the service station. In case no update is available, this field data will be forwarded to the OEM in order to allow a correlation of the field data provided by a representative number of products to allow the identification of possible software design faults (i.e. fleet analysis as engineering feedback).

5.9.3 Applying Out-of-Norm Assertions for Solving Prevalent Diagnostic Problems

ONAs can help to solve existing diagnostic problems of distributed embedded real-time systems. Among the most important problems are:

Reduction of the TNI phenomenon

The task of system diagnosis is to assess the operational state of a system. In case of a distributed system, diagnosis must operate on the distributed state to diagnose correlated errors and to indisputably judge about the functional correctness of the constituting parts of the systems, i.e. the components. In case the diagnosis subsystem operates on the local states, only those errors can be traced, that can be

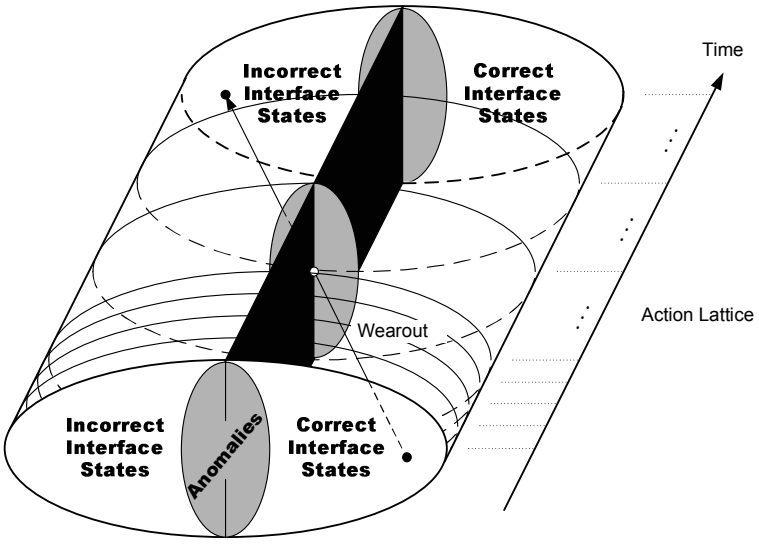

Figure 5.30: Assessment of a Wearout Phenomenon

classified without knowledge about other components. For instance, in the automotive domain the OBD systems tend to analyze local information in contrast to global information to assess the health condition of each ECU. In combination with a system architecture that provides only limited control of error propagation, such a diagnostic subsystem rather provides hints about possible faults than an exact identification of the component that needs to be replaced.

Consider for example transient internal versus transient external faults. Both types of faults are frequently causing spurious component failures in distributed embedded real-time systems. While an ONA covering transient internal faults restricts the space domain to only one FRU with respect to hardware faults, an ONA for the detection of transient external faults needs to cover multiple FRUs. This way the spatial proximity is taken into account, as a prerequisite for any analysis process. Since the symptoms of an ONA can be verified by multiple components by cross-checking the respective interface state, misleading error messages of faulty components can be identified and precluded from any further assessment.

Condition-Based Maintenance

TBM is increasingly being replaced by CBM, to reduce costs and to improve reliability and system performance [TS01]. Originally introduced to the avionics domain, this new paradigm is more and more accepted in the automotive industry. Besides the reduction of cost of ownership (service only what is needed) the possibility of collecting accurate field data (i.e. *engineering feedback*) is one of the major benefits from this maintenance approach. In addition, the customer trust in the car [Ber02] will be increased, since component replacements can be performed by the service technicians before the owner of the car is informed by the car's OBD system.

In order to adopt CBM for electronic systems suitable indicators for degradation or wearout must be identified and analyzed to detect deviations from sound operation. For example in machinery vibration, thermal, and lubricant analysis are good indicators for possible defective conditions [Sta97]. One of these indicators in the electronic domain is the increasing rate of transient failures [Con02].

ONAs can be deployed to detect deviations of components that indicate future component failure and transient errors resulting from intermittent type faults. The symptoms of ONAs are suited to detect those system states that are correct but at the verve of become incorrect in the near future. The continuous analysis of the acquired information by the ONAs allows for preventive measures at an early stage.

A wearout fault affects only a single FRU with respect to hardware faults and reoccurs repeatedly at the same location at higher rates with decreasing intervals [Con02]. Consequently, the time domain is of great significance in the identification of wearout phenomenons in order to realize CBM, whereas the spatial dimension is usually restricted to one FRU (see also the wearout fault pattern in Figure 5.9).

Consider for example clock synchronization. In case the correction term of the clock synchronization algorithm is larger than $\Pi/2$, where Π denotes the precision, the node raises a clock synchronization error and is excluded from the set of operational components. ONAs allow to detect such a drift (e.g., due to wear-out effects or rapid temperature change) at an early stage. In case the correction term is at the upper or lower limit of the correct range $[-\Pi/2, \Pi/2]$ a value anomaly with respect to the quartz has been identified. The evaluation process to identify wearout phenomenons is performed by the diagnostic subsystem and is illustrated in Figure 5.30.

The arrow in Figure 5.30 indicates this wearout phenomenon, i.e. the increasing drift rate of the quartz. At first the rate of the quartz is in conformance with its specification, but as time progresses, the quartz frequency drifts away from the specified frequency. This can be made obvious by measuring the clock correction term in case the system supports a global time base through clock synchronization. As soon as the correction term value is at the borders of the synchronization interval, an ONA is raised (i.e. this is a correct state). If the ONA fires repeatedly within an a priori defined interval, the diagnostic subsystem can conclude that the component will fail in the near future and indicate the maintenance engineer to change the component in time.

Exploiting the Three Dimensions

Integrated architecture provide a finer granularity of diagnostic information than federated systems. The decomposition of the overall system into DAS with respective jobs is a key element for a more precise differentiation of experienced faults. By including the three dimensions of time, value, and space into the judgment process, a discrimination into internal hardware faults, external hardware faults and software faults is possible.

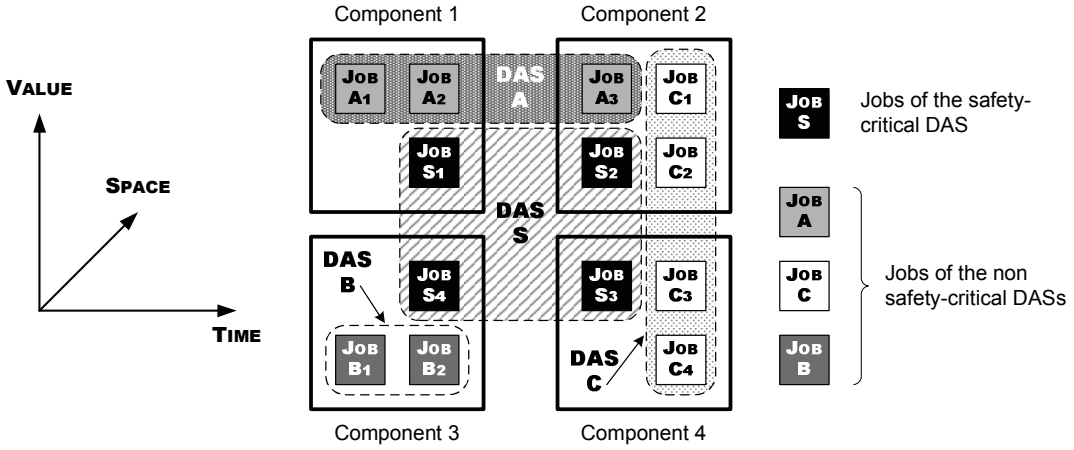

Figure 5.31: Judgment According to the Three Dimensions: Time, Value and Space

For instance, consider the system depicted in Figure 5.31. In case a job inherent fault hits the jobs A_1, A_2, and A_3 of the non safety-critical DAS A, the fault effects only the DAS A, since the error containment mechanisms of the architecture ensure that this fault cannot propagate to other DASs. In contrast, in case an internal component fault hits a component hosting multiple jobs of different DASs, it is very likely that the impact of this fault is not limited by DAS borders. An internal component hardware fault will cause multiple jobs hosted on one component to fail (e.g., the jobs A_3, C_1, C_2, and S_2 on component 2 in Figure 5.31).

The recognition of correlated job failures is also important in the detection of faults affecting architecture supported fault-tolerance mechanisms, such as TMR mechanisms. This fault-tolerance mechanism is characterized by the replication of identical jobs on three different components in order to tolerate single hardware faults. In case the jobs S_1, S_2, and S_3 are forming a TMR system, the spatial dimension of an ONA covering deviations in the services of the three replicas spreads across components 1, 2, and 3 (since a component is the FCR with respect to hardware faults). In case one of the replicated safety-critical jobs fails, an analysis if correlated failures of jobs of other DAS executed at the same time on the same component exist will supply evidence whether a an internal hardware fault effects the component.

In addition, for the differentiation whether transient failures are caused by environmental influences or internal faults, techniques such as the α-count mechanisms can be utilized [BCGG97]. By interpreting the detected failures in the time and space domain, a determination between external and internal component faults is possible, since transient component internal faults tend to occur at a higher rate compared to transient component external faults and occur repeatedly at the same location [Con02]. This discrimination is of paramount importance since internal component faults can only be eliminated by repair, while a replacement of a component due to an external component fault will only increase the fault-not-found ratio (i.e. the component will be retested OK at bench tests).

Chapter 6

Implementation of the DECOS Architecture and the Diagnostic Services

In the following we demonstrate how the introduced concepts have been implemented in a prototype setup of the DECOS integrated architecture. We first discuss the hardware and software setup of the prototype and the design rationale for the deployed DASs. Furthermore, we briefly describe the jobs for each DAS and the corresponding configuration of the virtual network service.

In this chapter we also elaborate on the framework provided by the DECOS architectural services to integrate diagnosis as a central part of the architecture. The use of timed automata for the specification of ONAs allows automatic code generation of both the symptom detectors and analysis algorithms with precisely defined execution semantics. Based on the introduced maintenance-oriented fault model we exemplify the use of the provided framework for the specification and execution of systemic and application-specific symptoms. In particular, we elaborate on the information provided by the TTP controller that can be used for the detection of systemic failures. In addition, we discuss the design and implementation of the analysis job.

Since the DECOS architecture, as part of the Sixth European Framework Programme, is still under development, the validation of the high level services like the virtual network and inner-component partitioning mechanisms is still in progress. For preliminary performance measurements of the used analysis algorithms we assume that the exploited high level services are free of design faults. Based on this assumption we are able to discuss the performance of selected out-of-norm analysis algorithms or determining the faulty FRU according to the maintenance-oriented fault model. The presented results primarily show the feasibility of the chosen strategy, of operating on the distributed state instead of the node local state, to improve the accuracy of the online diagnostic solution.

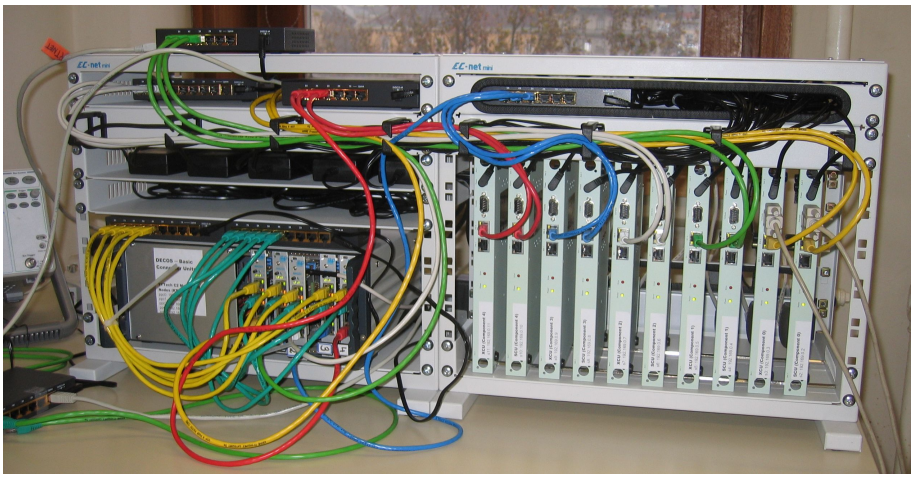

Figure 6.1: The DECOS Prototype Cluster

6.1 The DECOS Architecture Prototype Setup

Our prototype implementation of the DECOS architecture consists of a cluster of five components using TTP/C [Kop99b] as the time-triggered core communication service. Each component hosts a safety-critical and a non safety-critical subsystem with multiple DASs and corresponding jobs. The prototype cluster is depicted in Figure 6.1.

6.1.1 Hardware Setup

For the implementation of a DECOS component according to the model presented in Section 4.4, we employ distinct hardware elements for the BCU, the safety-critical subsystem, and the non safety-critical subsystem (i.e. one node computer for each SCU/XCU and respective applications). Since we share the same computational resources between the secondary connector unit (i.e. SCU and XCU) and the encapsulated execution environment of the jobs, the need for inner-component error containment by means of temporal and spatial partitioning is raised in order to protect architectural services from job interference on the one hand, and interference between jobs on the other hand.

The interconnection between the BCU and the secondary connector units is realized using time-triggered Ethernet. At a priori defined points in time, an Ethernet message containing the state information that is disseminated on the core network is transferred from the BCU to the secondary connector units and vice versa.

Basic Connector Unit

In our prototype implementation we deploy the TTTech[1] monitoring node [TTT02a] as the basic connector unit establishing the time-triggered core communication service (see Figure 6.3). The TTP monitoring nodes are based on the TTP-C2 controller (AS8202). They are equipped with the Freescale embedded PowerPC processor MPC855T.

In integrated architectures a physical component is shared among multiple jobs. Therefore, it is important to decouple the capability of a node to transmit messages on the physical time-triggered core network from the functionality of the hosted jobs. In case only a minority of jobs (i.e. software modules) hosted in their respective partitions fail, it still must be possible for other jobs to utilize the architectural services and to communicate via their dedicated virtual network. Note, that according to the DECOS fault hypothesis, the complete node computer is considered to be a FCR with respect to hardware faults [KOPS04].

In the DECOS architecture the decoupling is realized via the BCU. The BCU guarantees message exchange via the core network by not relying on the information provided by the secondary connector units. Such a decoupling is only possible in case a state message interface is provided [Kop97].

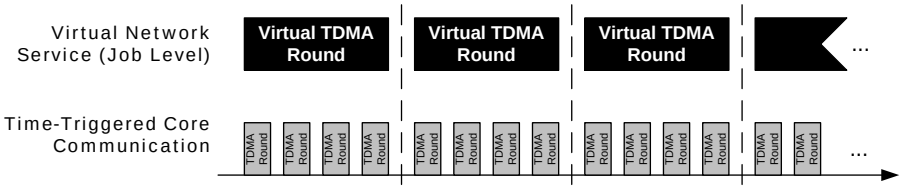

Figure 6.2: The Independence of the Core Communication System from the Applications

This independence of the time-triggered core communication system from the jobs facilitates a decoupling of the TDMA schedule of the core network from the schedule required by the applications. This means, that the TDMA communication schedule (i.e. the cluster cycle) of the physical time-triggered core network can be significantly shorter than the application requirements in order to provide higher bandwidths. The latency, however, is still limited by the TDMA schedule required by the high level services in order to provide the virtual network service for each DAS. For instance consider Figure 6.2. Here, the available bandwidth for the virtual network service is quadrupled transparently to the applications.

Secondary Connector Units and Application Computers

For the secondary connector units we use the Soekris Engineering[2] net4521 embedded system as our target computer. This compact computer (depicted in Fig-

[1] `http://www.tttech.com`
[2] `http://www.soekris.com`

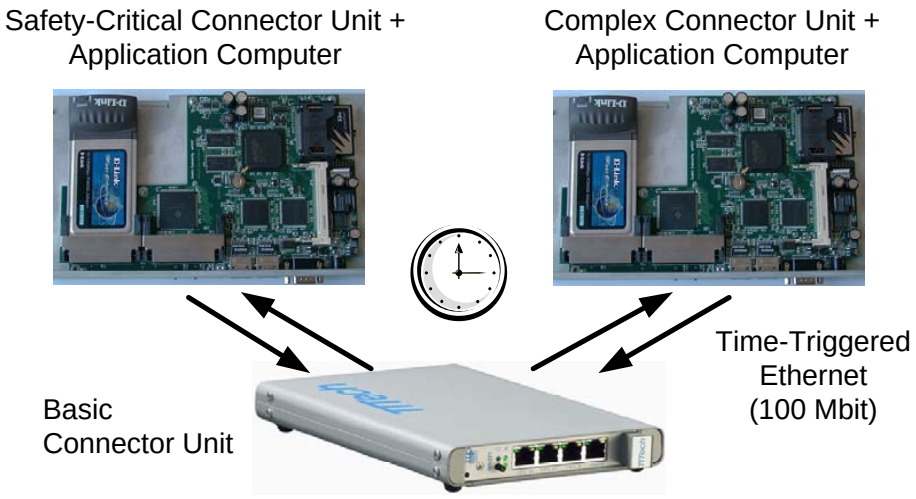

Figure 6.3: A Prototype DECOS Component

ure 6.3) is based on a 133 Mhz 486 class ElanSC520 processor from AMD. It has two 10/100 Mbit Ethernet ports, up to 64 Mbyte SDRAM main memory and uses a CompactFlash module for program and data storage. Furthermore, it has two PC-Card/Cardbus adapters which allow an easy extension of the system.

6.1.2 Software Setup

Both, the BCU and the secondary connector units use the embedded real-time Linux variant Real-Time Application Interface (RTAI) [BBD+00, RTA00] as their operating system. While in the BCU only kernel modules are used, in the secondary connector units the partitioning capabilities of the RTAI/LXRT extension are heavily utilized.

The software configuration used for the prototype implementation combines the ADEOS[3] hardware abstraction layer with a real-time application interface for making Linux suitable for hard real-time applications. In the prototype setup we use RTAI v3.1 on a Linux 2.6.9 kernel (including the real-time RTnet Ethernet driver suite). RTAI introduces a real-time scheduler, which runs the conventional Linux operating system kernel as the idle task, i.e. non real-time Linux only executes when there are no real-time tasks to run and the real-time kernel is inactive. The conventional Linux kernel is modified to prevent it from blocking or redirecting hardware interrupts. Hence, Linux cannot add latency to the interrupt response time of the real-time system. RTAI performs these modifications by replacing the corresponding code in the Linux kernel with calls to functions in the real-time hardware abstraction layer. This mechanism offers the possibility for software emulation of interrupt control hardware. When an interrupt occurs, the real-time kernel intercepts the interrupt and runs its own dispatcher, which invokes the corresponding real-time handler. In order to prevent temporal fault propagation from the Linux kernel to the real-time kernel, the real-time kernel is never blocked by the Linux side.

[3] `http://www.adeos.org`

Linux Real-Time (LXRT) is an extension of RTAI that enables the development of hard-real time programs running in user space instead of kernel space. Furthermore, LXRT eases the communication between hard real-time and non real-time processes, which can be utilized for monitoring and debugging of the hard-real time processes, without affecting their real-time behavior.

Temporal and Spatial Partitioning using RTAI/LXRT

As elaborated in [Rus99], the purpose of partitioning is to ensure that the execution of a job hosted in one partition is not affected by a job in another partition. Thus, partitioning has to inhibit the propagation of software failures between jobs. Since in DECOS a job is regarded as FCR for software faults (according to the fault hypothesis introduced in Section 4.6) and the secondary connector units and jobs share the same computational resources, means must be deployed that establish temporal and spatial partitioning in our prototype implementation.

- **Spatial Partitioning.** Spatial partitioning needs to prevent jobs from overwriting memory elements of other jobs and preventing jobs in interfering in the access of devices [Rus99]. Unlike to real-time applications using the RTAI Application Programming Interface (API), which are realized as Linux kernel modules and executed in supervisor mode, real-time applications utilizing the LXRT extension of RTAI retain in user mode. Since LXRT allows the execution of hard real-time jobs in user mode, it preserves the memory protection mechanisms of the Linux operating system and provides support for spatial partitioning (i.e. memory access is protected by Memory Management Unit (MMU) tables in user mode). Furthermore, since jobs in DECOS have exclusive access to I/O, explicit synchronization mechanisms for protecting devices are not required.

- **Temporal Partitioning.** The purpose of temporal partitioning on the other hand is the retention of a correct schedule even in the case of faulty jobs holding a shared resource (e.g., the processor). Since we favor predictability and simplicity to flexibility and better support for sporadic tasks, in the prototype implementation a static job schedule that is generated off-line during the system integration phase. Similar considerations as presented in [LKYZ00] have to be taken into account when developing a feasible schedule. The job schedule is synchronized to the TDMA schedule of the underlying core communication service. It forms the basis of our execution environment and consists of a list of jobs together with their execution times that are to be executed in the respective TDMA slot. See Figure 6.4 for a sample schedule. Since RTAI lacks the possibility to specify a deadline or a maximum execution time until a real-time process has to have finished its execution, we have developed a time-triggered dispatcher extending the RTAI/LXRT functionality that activates high-level services and jobs according to a static timetable and ensures that

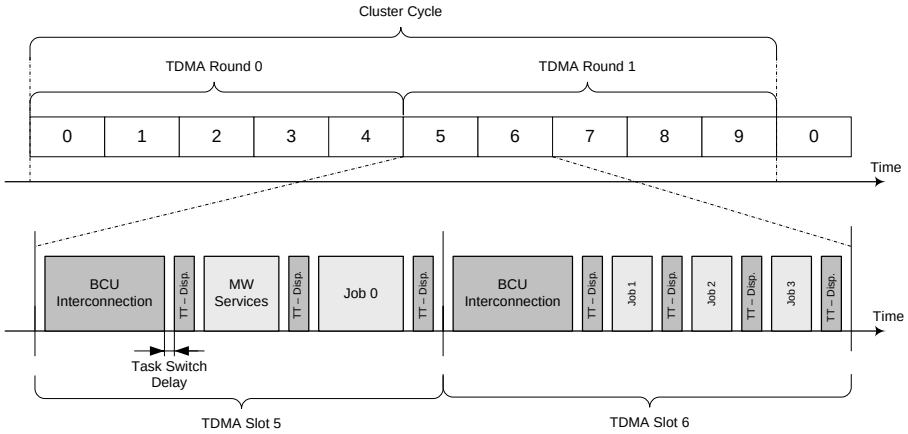

Figure 6.4: BCU Interconnection Schedule for Node 0 of the Cluster

their assigned execution times are not exceeded. For a detailed discussion of the time-triggered dispatcher see [HPOS05].

Like error containment at network level is a prerequisite for precise diagnosis, inner component error containment by means of partitioning is important not only to allow independent development of jobs and DASs, but also to identify those jobs that are responsible in case of failure.

6.1.3 Hierarchical Network - Interconnection Scheme

As stated, the DECOS architecture supports the decoupling between the communication system and the applications via the establishment of connector units. In fact, the BCU of one component exchanges information with other components regardless of the temporal accuracy of the data stored in the state message interface.

In order to forward the information received from other components, the BCU sends the respective data to the secondary connector units. In the prototype implementation the BCU and the XCU/SCU of a component are interconnected by a time-triggered Ethernet link. This way a hierarchical time-triggered network is realized that significantly reduces the complexity and due to the static nature allows simplified error detection. An exemplary interconnection schedule for component 0 of the DECOS prototype cluster is presented in Figure 6.5. In each TDMA slot of the underlying time-triggered protocol (i.e. TTP/C in the prototype implementation) the data of the last TDMA slot is transferred to the secondary connector units. Each connector unit has a dedicated slot in which the data of the applications associated with the respective connector units is sent back to the BCU for dissemination on the time-triggered core network.

In the implementation the data and control/status areas of the core communication controller (i.e. the TTP C2 controller [TTT02b]) are mapped into the respective data structure of the secondary connector units by the allocation layer. All higher

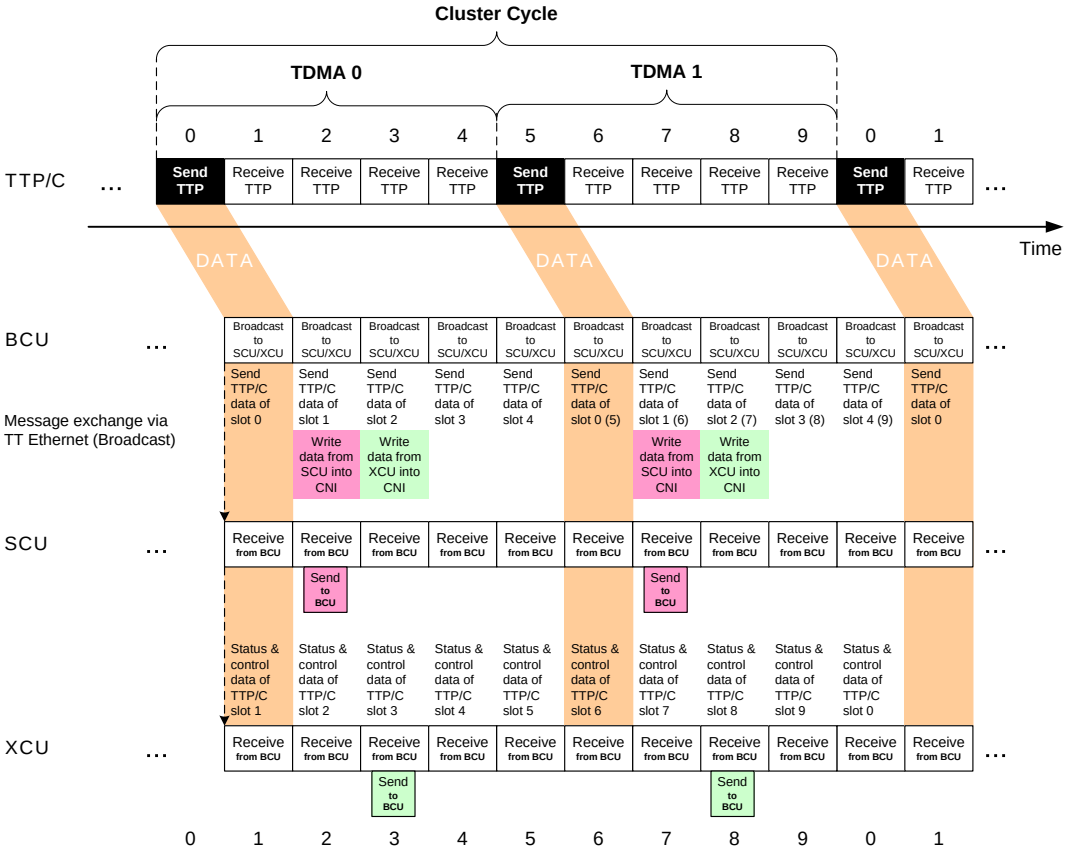

Figure 6.5: Hierarchical Networks: Interconnection Schedule for Node 0 of the Cluster

layers as introduced in Section 5.8 operate on and update the elements of this central data structure.

6.1.4 Automotive Example

In order to show the suitability of the DECOS architecture for automotive applications [POT+05], in the prototype setup a hypothetical automotive infrastructure is realized with the following DASs, respective jobs, and virtual networks:

- **Drive-by-wire DAS.** The drive-by-wire DAS employs a time-triggered virtual network that interconnects the jobs controlling the gear box, the engine, the brakes, and sensing the input values from the steering wheel, and pedals of the car.
- **Comfort DAS.** The jobs of the comfort DAS control the passenger compartment of the car and communicate via an event-triggered virtual network. Among the services provided by the jobs of this DAS are the air conditioning, passenger and driver door functionality, the sliding roof, and trunk lid control.

- **Navigation DAS.** The navigation DAS is an example for a non safety-critical DAS that is implemented via a time-triggered virtual networks. The state semantics of the time-triggered virtual network is especially useful for the dissemination of the Global Positioning System (GPS) data.
- **Lights DAS.** This DAS is semi-virtual, i.e. it interconnects a virtual network with a physical one. While the actual controlling of the exterior car lights (including adaptive forward lighting) is realized via CAN nodes, a part of the functionality of the network is realized via event-triggered jobs. This DAS is used to demonstrate the seamless integration of physical fieldbusses into a DECOS cluster.

Naturally, this system structure calls for the exchange of information between the DASs, e.g., the steering angle is forwarded to the lights DAS for the adaptive forward lighting. Therefore, the high-level gateway service is heavily exploited.

In the prototype setup a maximum of six jobs and the respective high level services are hosted on each physical DECOS component. According to the schedule discussed in Section 6.1.2, three jobs are executed in each subsystem (safety-critical and non safety-critical).

From a diagnostic perspective, the virtual diagnostic network and the deployment of the diagnostic DAS are of special interest. As described in Section 5.6 the virtual diagnostic network comprises the ALD channel and the channels for the transportation of job internal symptoms (i.e. diagnostic information not accessible via the interface state due to IP issues). Since the implementation of the virtual network service in case of event-triggered communication allows fragmentation of messages, the system designer has the choice between short latencies of the diagnostic messages or scarce allocation of available bandwidth. This tradeoff has been solved in our prototype setup by allocating the majority of the bandwidth reserved for diagnosis to ALD, since this information results from applying the cross-checking principle introduced in Section 5.5.2.

As a result of this design seven virtual networks are realized in the prototype cluster and the following identifications are used in the prototype implementation:

```
#define NETWORK_CAN_COMFORT     0 // event-triggered VN
#define NETWORK_AVDN_NSC        1 // event-triggered VN (diagnosis)
#define NETWORK_PVDN_NSC        2 // event-triggered VN (diagnosis)
#define NETWORK_PVDN_SC         3 // time-triggered VN (diagnosis)
#define NETWORK_CAN_LIGHTS      4 // event-triggered VN
#define NETWORK_TT_NAVIGATION   5 // time-triggered VN
#define NETWORK_TT_BYWIRE       6 // time-triggered VN
```

6.1.5 Implementation of the Virtual Network Service

Since the communication in the DECOS cluster – not only for diagnostic purposes – relies on the virtual network service, we will elaborately briefly on the realization of

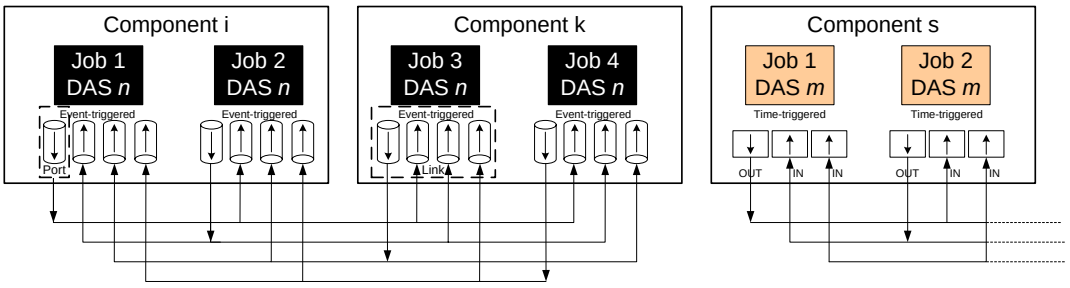

Figure 6.6: Implementation of the Virtual Network Service

this service in the prototype implementation.

According to the concepts introduced in Section 4.3.2, each DAS has a virtual network as the communication infrastructure tailored to the requirements of the application domain as depicted in Figure 6.6. Each virtual network specification in the DECOS cluster contains the name of the network, provides information of whether the network adheres to either the event-triggered or time-triggered paradigm, and has a specified number of corresponding links.

```
typedef struct vn_struct {
  char                 *name;
  enum vn_paradigm     paradigm;
  int                  number_of_links;
  t_link               link[MAX_LINKS_PER_VN];
} t_vn;
```

A *link* is the interface for each job to the virtual network service. Such a link contains information about the job it belongs to, the physical component and subsystem the job is hosted on, the number of ports comprising the link, and references to each port.

```
typedef struct link_struct {
  t_job                *job;             // the job the link belongs to
  unsigned short       component;        // the component and subsystem
  unsigned short       subsystem;        // at which the link is located
  int                  number_of_ports;  // number of ports in the link
  t_port               port[MAX_PORTS_PER_LINK];
} t_link;
```

Depending on the paradigm of the virtual network (event-triggered vs. time-triggered), the port is either realized as a queue or a state variable. Furthermore, each port has a direction (i.e. input or output), and in case of an event-triggered port an indication whether an overflow has occurred. Besides these configuration data, each `t_port` data structure contains additional information for the packet service. The packet service provides message fragmentation functionality to allow effective use of available bandwidth.

```
typedef struct port_struct {
  int                  q_len;    // ==1 for TT communication, >=1 for ET
  int                  msg_len;  // max. length of the message in bytes
  enum direction       dir;      // data direction
  enum vn_paradigm     paradigm; // ET or TT
  t_msg_descr          *msg;     // reference to msg information structure
  t_port_buffer        *buf;
  struct link_struct   *link;    // backward reference to the link
  unsigned char        overflow; // 1 == overflow
  struct vn_struct     *vn;      // the virtual network the port belongs to
  ...
  // Information for the packet service
  ...
} t_port;
```

Each job of the DECOS cluster can easily access the corresponding virtual network by connecting to its link during the initialization phase at startup. By calling the `fetch_link()` function, the connection to the virtual network is established. For example, the following code snippet shows how a job establishes the access to the virtual network with the identification `NETWORK_CAN_COMFORT`.

```
static t_link *Link2CAN_COMFORT;
...
if ((Link2CAN_COMFORT = fetch_link(NETWORK_CAN_COMFORT,JOB0)) == NULL) {
    rtai_print_to_screen("Error fetching NETWORK_CAN_COMFORT link\n!");
    return -1;
}
```

The second parameter specifies the identification of the job that wants to access its link. `fetch_link()` catches the shared memory region that maps the interface of the virtual network service in the user space where the LXRT task can access its encapsulated communication resources.

In case of an event-triggered job, the job can now send a message whenever a change in an interface state variable occurs. Therefore, the job sets the length of the message and updates the content of the message according to the new values.

```
t_message ET_msg;
...
CAN_msg.len = 4;
ET_msg.data[0] = sensor1_16();  // Temperature sensor measuring
ET_msg.data[1] = sensor1_8();   // the outside temperature
ET_msg.data[2] = sensor2_16();  // Temperature sensor measuring
ET_msg.data[3] = sensor2_8();   // the passenger compartment temperature
...
sndEtMessage(&(Link2CAN_COMFORT->port[0]), (t_message*) &ET_msg);
```

Once the message is constructed, the job calls `sndEtMessage()` with the parameters `t_port *port` for specifying the output port and `t_message *msg` for providing a pointer to the outgoing message. In analogy, a job may received messages by calling the `rcvEtMessage(..)` function with according parameters such as the input port.

In case of a time-triggered virtual network instead of accessing the queue, a state variable is periodically updated or read respectively according to the TDMA schedule of the virtual time-triggered network.

6.2 The Diagnostic Framework

In order to integrate diagnosis into the development process, a framework that supports both, the application developers and system designers is needed. Such a framework also forces the developers to precisely specify the diagnostic checks and to treat diagnosis not as an addendum but as an integral part of all development phases. By applying a framework, additional design faults can be avoided by providing the possibility to automatically transform the symptom detectors and analysis automata comprising the ONAs specifications into executable code that can be executed as part of the high-level services. Figure 6.7 gives an overview on the proposed framework.

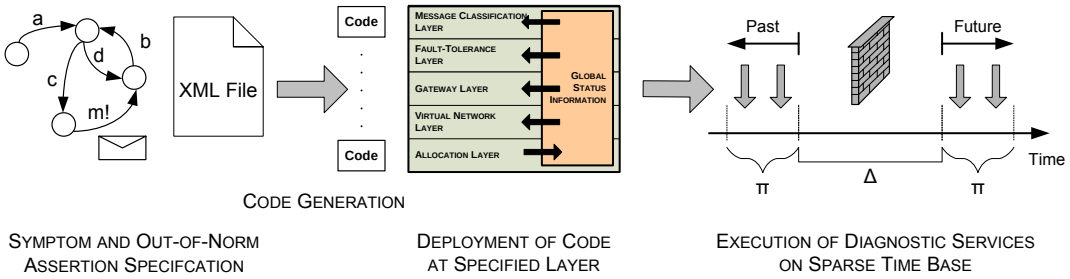

Figure 6.7: The Diagnostic Framework

6.2.1 Specification of Out-of-Norm Assertions

As proposed in Section 5.7 for the specification of the diagnostic checks timed automata are used. By constraining the types of automata that can be used for specification to the subclass of deterministic and nonzeno timed automata (cf. Section 5.7), it is assured that these automata can automatically be compiled into an executable version and executed as part of the diagnostic middleware. By using a tool with graphical user interface like the frontend for the Uppaal model checker [PL00] the specification of diagnostic checks is simplified.

6.2.2 Code Generation

Based on a formally expressed specification, executable code according to target platform specific parameters can be generated. A compiler can then transform these

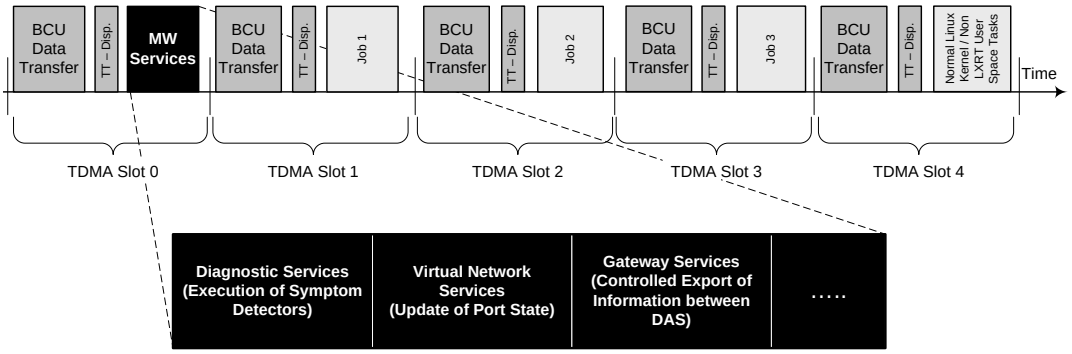

Figure 6.8: Schedule of the Diagnostic Middleware Services

specifications into executable code. For example, a formal specification for timed automata expressed in XML is presented in [OP05] in the context of gateways. In order to allow execution of the diagnostic automata introduced in this thesis according to the semantics as defined in Section 5.7.4, the variables used in the definition of guards and actions of the specified automata need to be replaced by respective port state variables. All specified receive or send operations need to be parameterized according to the configuration of the virtual networks as a result of the cluster configuration activities. Furthermore, the compiler needs to substitute all timing parameters expressed in standard physical units (e.g., seconds, ms, μs) with macroticks according to the schedule of the time-trigged core communication protocol. Automatic code generation has the significant advantage of avoiding coding errors that can significantly affect the diagnostic results during operation. In addition, once defined diagnostic mechanisms, i.e. symptoms and ONAs, can be reused.

6.2.3 Deployment and Execution

During the initialization phase of the DECOS cluster, the diagnostic middleware processes the cluster configuration data structures and determines the allocation of jobs to the physical components. According to the cluster configuration, the diagnostic middleware connects to the virtual diagnostic networks and connects to the respective ports of the jobs hosted on each component. This is necessary to allow monitoring of all incoming and outgoing messages at the virtual network interface.

As depicted in Figure 6.8 the diagnostic middleware executing the symptom detectors is scheduled according to the TDMA schedule of the underlying time-triggered core network (see also Section 6.1.2). At the beginning of each TDMA round the diagnostic checks are executed. This is sufficient, since each physical component is allowed to send only once during a TDMA round on the TTP core communication network.

The interconnection between the BCU and secondary connector units basically mirrors the CNI of the underlying time-triggered communication controller (in this case the TTP C2 controller [TTT02a, TTT02b]) into the memory of the secondary

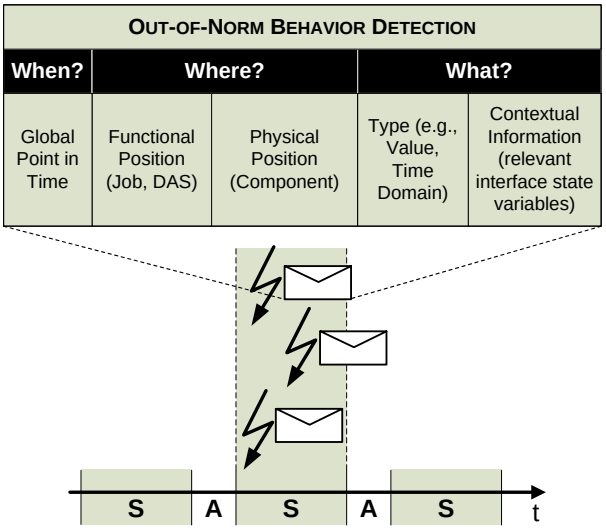

Figure 6.9: Execution on the Sparse Time Base

connector units. In our prototype implementation, this data structure is called `exchange` and stores for each slot of the TDMA schedule the following information:

```
struct slot_type{
  unsigned char status[CNI_STATUS]; // Status Area
  unsigned char ctrl[CNI_CTRL];     // Control Area
  unsigned char ch0[CNI_CH0];       // Data Channel0
  unsigned char ch1[CNI_CH1];       // Data Channel1
};
```

The diagnostic middleware has not only full access to all relevant status information of the underlying time-triggered core network stored in the `exchange` data structure, but also to the data structures of all other high-level services (e.g., the virtual network data structures as introduced in Section 6.1.5).

In Section 5.8 the layers of a DECOS component have been described. Emphasis has been set on the types of diagnostic checks that can be performed at each layer. As depicted in Figure 6.9 the timed automata are executed in each silence interval (denoted S) of the virtual network service. This ensures that a system wide consistent view on the interface state of each component/job has been established and thus guarantees location transparency of the deployed checks. Whenever a violation of an interface specification has been detected, a corresponding diagnostic message is constructed and pushed into the outgoing port (i.e. queue) of the virtual diagnostic network.

Systemic Symptom Execution

As indicated in Figure 6.10 the systemic symptom detector is independent of any job-specific application logic. This enables reuse of the specified checks in a large variety

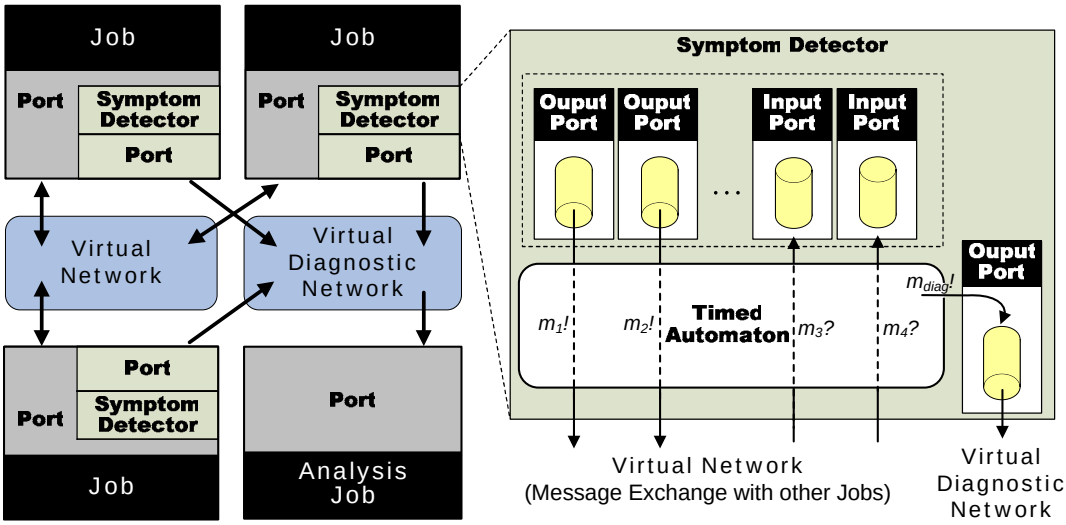

Figure 6.10: Symptom Detection

of systems in case the symptom has proven in the field to provide valuable diagnostic information. Systemic diagnostic checks do not need to include any knowledge about the physical or functional structure of the system. This is solely encoded into the ONA processing the information provided by symptoms.

The timed automata encoding the symptoms are executed on the action lattice of the sparse time base according to the algorithm introduced in Section 5.7.4. As long as a transition can be taken, the timed automata for symptom detection are executed, otherwise the execution is stopped until the next interval of silence. This way correlated symptom detection is ensured and there is no need for the execution of explicit agreement protocols on symptoms. Furthermore, for the analysis process we can exploit this property of the sparse time base and can be sure that messages with different timestamps encode different diagnostic events. The only exception might arise for faults affecting the state for an interval of time that is longer than the respective interval on sparse time base (e.g., heavy EMI disturbances).

Furthermore, the location transparency property allows deploying the symptoms at different physical locations. Since symptoms are defined on the interface state (i.e. on the content of messages) and the interface state is exchanged via broadcast communication to all communication partners, the cross-checking principle can be realized for a majority of diagnostic checks. This way, not only trustworthy information is generated, but also available resources can be used very efficient.

For systemic checks, the `exchange` data structure is of central interest, since it provides the controller information of the underlying TTP cluster. In case a hardware failure has been detected by the controller, an entry in the respective control register is performed. The value of this control register (e.g., the clock state correction term, frame status field) can now easily be accessed by the symptom detector to check for out-of-norm values.

Application-Specific Symptom Execution

The job-specific symptom detectors are hosted at the message classification layer. The diagnostic checks (e.g., plausibility checks for sensor values) are evaluated against the messages traversing the input and output ports of a particular job. Once a message violates the port specification of the sending or receiving job a corresponding diagnostic message is constructed and transferred via the virtual diagnostic network. Similar to the systemic symptom detection the timed automata encoding application specific diagnostic checks are executed on a sparse time base provided via the underlying core time-triggered network. In contrast to systemic checks that are periodically executed, the execution of symptoms for event-triggered jobs depends on the availability of messages in the ports (i.e. the queues).

As already stated, during system startup the diagnostic services determine which jobs are running on each physical component. According to the configuration data for each hosted job on the component the diagnostic middleware processes the `t_application_check` data structure that contains for each job one or more corresponding function pointers that implement the application-specific symptom detector.

```
t_application_check execute_symptoms[MAX_APP_SYM] = {
  {JOB0_COMFORT,job0_comfort_no1},  // "Air Condition Front"
  {JOB0_COMFORT,job0_comfort_no2},
  {JOB0_COMFORT,job0_comfort_no3},
  ...
  {JOB5_BYWIRE,job5_bywire_no1},    // "Gas/Brake Sensor 0"
  {JOB5_BYWIRE,job5_bywire_no2},
  {JOB6_BYWIRE,job6_bywire_no1},    // "Steering Sensor 0"
  {JOB6_BYWIRE,job6_bywire_no2}
};
```

For instance, the function `job0_comfort_no1` contains the code of a symptom detector specified as a timed automaton for the monitoring of a temperature sensor value. The structure of such a typical check for an output port of an event-triggered job is as follows:

```
void job0_comfort_no1(t_port *port, t_diag_msg *diag_msg,....) {
    t_message *SourceMessage;
    char      *pSourceArea;
    ...
    wr_pos = port->buf->wr_pos;
    rd_pos = port->buf->rd_pos;
    ...
    // Check for new messages in the queue
    if (last_wr_pos != wr_pos)) { //av(m)?
         while (rd_pos != wr_pos) { // read all messages (non consuming)
```

```
            pSourceArea   = ((char*) &(port->buf->msg)) +
                              (rd_pos*(port->msg_len+ETSERVICE_HDR_SIZE));
            SourceMessage = (t_message*) (pSourceArea);
            ...
            // Check on message content
            if ((SourceMessage->data[0] < ...) ... ){
                  // construct diagnostic message
                  diag_msg->len = 7;
                  diag_msg->data[0] = TYPE_APP;
                  diag_msg->data[1] = APPSYM_SENSOR_AT_THRESHOLD;
                  ...
                  diag_msg->data[6] = SourceMessage->data[0];
            }
            rd_pos++;   // updating read pos; no effect on VN service!
            if (rd_pos>=port->q_len) rd_pos=0; // limited queue size
         }
         last_wr_pos = wr_pos;   // update last position
    }
}
```

Note, that in case no message is available, the execution of the automaton is suspended. This also applies in case additional timing constraints (like monitoring of minimum interarrival times between messages) are specified.

6.3 Implementation of the Symptoms

In the following we describe specification and implementation of the deployed symptom detectors for the identification of faults according to the maintenance-oriented fault model introduced in Section 5.3. We show how these systemic and application-specific checks are realized in the prototype system. For each class of the maintenance-oriented fault model we exemplify the timed automaton encoding the assertion on the interface state on the basis of a real-world example. Especially, we elaborate on the information provided by the TTP C2 controller that can be used for accurate diagnosis of hardware failures.

6.3.1 Job-Inherent Faults

The monitoring of the interface state of applications is important to detect job-inherent faults, i.e. software faults or faults induced due to faulty sensors or actuators connected to the job. In the following we demonstrate how the job-inherent symptom detection is realized in the DECOS architecture on the basis of both a time-triggered and an even-trigged job example.

Figure 6.11: Steering Wheel and Pedals

Time-Triggered Job

In the prototype setup the drive-by-wire DAS consists of a number of time-trigged jobs providing basic car functionality, such as engine, brake and steering control. For simulating the driver's input a Logitech WingMan Formula GP USB steering wheel with corresponding pedals has been connected to one of the Soekris nodes (using a D-Link USB DUB-C2 card). As depicted in Figure 6.11 the human interface device provides the following input values:

- Both the brake pedal force and gas pedal force can be directly read via the USB interface. For simplicity each value has a resolution of 255 steps. Consequently a two byte message is needed for sending the current pedal forces.
- As for the pedal values, the steering wheel angle is discretized into a byte value, allowing a resolution of 127 steps for each driving direction. In addition, the user can activate six different buttons on the steering wheel. In the prototype setup two buttons are used for the gearshifts (up and down), for the turn signals, and for temperature control (see also Figure 6.11). A one byte variable (of type `unsigned char`) is used for the encoding of this state information.

In our prototype setup one job measures the pedal forces applied by the driver to each pedal and updates the relevant interface state variables that are periodically disseminated on the virtual time-triggered network of the by-wire DAS. In analogy, another job determines the steering wheel angle and status information of each button of the steering wheel and distributes this information periodically.

In the diagnostic framework of the prototype system the diagnostic middleware executes periodically (at TDMA slot or round granularity) the symptom detection mechanisms. For each job one or more symptoms are defined in a function which is

called through a function pointer. A typical symptom detector for a time-triggered job in pseudo-code notation looks as follows

```
void job5_bywire_symptom1(t_port *port, ...., t_diag_msg *msg) {
    ...
    // read port state variables
    state_variable = port->buf;
    ...
    // check values of the port state variables
    if (state_variable == ...) {
        ...
    }
    ...
    // Construct diagnostic message if necessary
    if (something_wrong) {
      msg->data[0] = TYPE_APPLICATION;
      msg->data[1] = APPLICATION_SYMPTOM;
      msg->data[2] = GLOBAL_TIME1;
      msg->data[3] = GLOBAL_TIME2;
      msg->data[4] = NETWORK_TT_BYWIRE;
      msg->data[5] = JOB5;
      msg->data[6] = state_variable;
    }
}
```

Since in most driving situations, either the brake or the gas pedal is exclusively pressed, a symptom indicating possible sensor malfunction monitors simultaneous operation of both pedals. In case the brake pedal value is pressed, then the applied gas pedal force must not exceed a predefined threshold value and vice versa. Otherwise a job-inherent symptom is detected and a corresponding diagnostic message as described in Table 6.1 is forwarded to the diagnostic DAS that can correlate this information with other job-inherent failure messages originating from the same job or from the component hosting the job to trace correlated failures and update the frequency of occurrence. The corresponding timed automaton for the specification of this symptom is shown in Figure 6.12. By transforming the timed automata into executable code, the variable names have to be replaced by statements that read the respective state variables (`job5_data[0]` and `job5_data[1]`) and the threshold values T_{Brake} and T_{Gas} defined accordingly.

```
if ((job5_data[0] > GAS_LIMIT) && (job5_data[1] > BRAKE_LIMIT)) {
    // construct diagnostic message
    ....
}
```

In analogy to the symptom detectors defined for the pedal values, similar checks can be performed for evaluating the plausibility of the steering wheel angle and

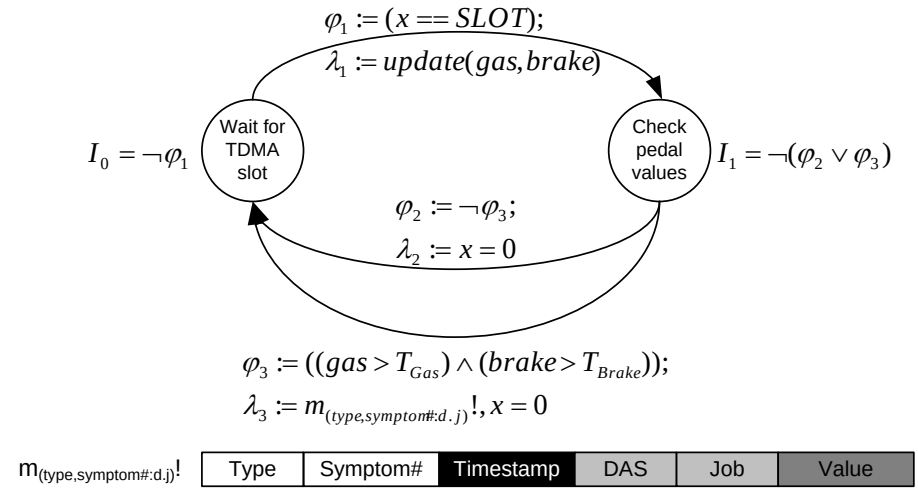

Figure 6.12: Timed Automaton for Symptom: *Both Pedals Pressed*

Field Name	**Semantics**
Message Type	This field is set to `TYPE_APPLICATION` in order to differentiate the message from systemic symptoms.
Symptom ID	The unique symptom ID is set to `APPSYM_BOTH_PEDALS_PRESSED`
Time Domain	This field includes timestamp of the global time base.
Space Domain	The DAS value of set to `NETWORK_TT_BYWIRE` and the job indication field to `JOB5`.
Value Domain	Values of the measured brake pedal and gas pedal force.

Table 6.1: Diagnostic Message for Out-of-Norm Pedal Forces

button values. Many cars of today offer a multifunction steering wheel with cruise control, audio, and multifunction onboard computer functions. By assigning each of the available buttons of the Logitech steering wheel a specific functionality such a multifunction steering wheel is realized in the prototype setup. From a diagnostic perspective some combinations of simultaneous pressed buttons do not make any sense, and thus are classified as out-of-norm and once detected forwarded to the diagnostic DAS. In the prototype implementation two switches alters the gearshift, two buttons implement blinker functionality, and two buttons are used for temperature adjustment. The automaton for the button specific checks is depicted in Figure 6.13. Every TDMA slot the state values are evaluated against the guards of the timed automaton and in case one of the guards φ_3, φ_4 or φ_5 are enabled a diagnostic message is created (see action λ). For a detailed layout of the fields of the diagnostic message refer to Table 6.2. The corresponding executable version of this symptom detector is as follows:

```
// both gear buttons simultaneously
if ((job6_data[1] & 0x03) == 0x03) {
    msg->data[1] = APPSYM_BOTH_GEARS_PRESSED;
    ...
```

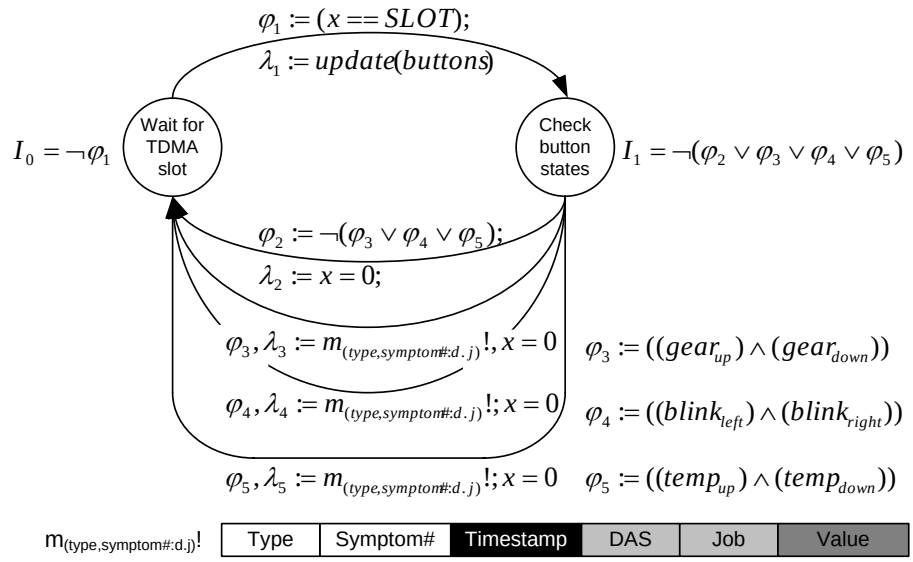

Figure 6.13: Timed Automaton for Symptom: *Out-of-Norm Buttons*

Field Name	**Semantics**
Message Type	This field is set to `TYPE_APPLICATION` in order to differentiate the message from systemic symptoms.
Symptom ID	The unique symptom ID is set to `APPSYM_BOTH_GEARS_PRESSED`, `APPSYM_BOTH_BLINKER_PRESSED`, `APPSYM_BOTH_TEMP_PRESSED`, or `APPSYM_ALL_BUTTONS_PRESSED`, depending on the detected out-of-norm condition.
Time Domain	This field includes timestamp of the global time base.
Space Domain	The DAS value of set to `NETWORK_TT_BYWIRE` and the job indication field to `JOB6`.
Value Domain	The status of the steering wheel buttons.

Table 6.2: Diagnostic Message for Out-of-Norm Multifunction Steering Wheel Inputs

```
}
// both blinker buttons simultaneously
if ((job6_data[1] & 0x0C) == 0x0C) {
    msg->data[1] = APPSYM_BOTH_BLINKER_PRESSED;
    ...
}
// both temperature buttons simultaneously
if ((job6_data[1] & 0x30) == 0x30) {
    msg->data[1] = APPSYM_BOTH_TEMP_PRESSED;
    ...
}
```

A nice example for monitoring the compliance of measured sensor values with physics laws is the observation of the changes of the steering wheel angle that is

sampled each TDMA slot of the virtual time-triggered network. In case the changes in the steering wheel angle are highly unlikely or impossible according to the underlying physics model a symptom has been detected. The timed automaton for this diagnostic check is shown in Figure 6.14. Γ denotes the maximum value that the

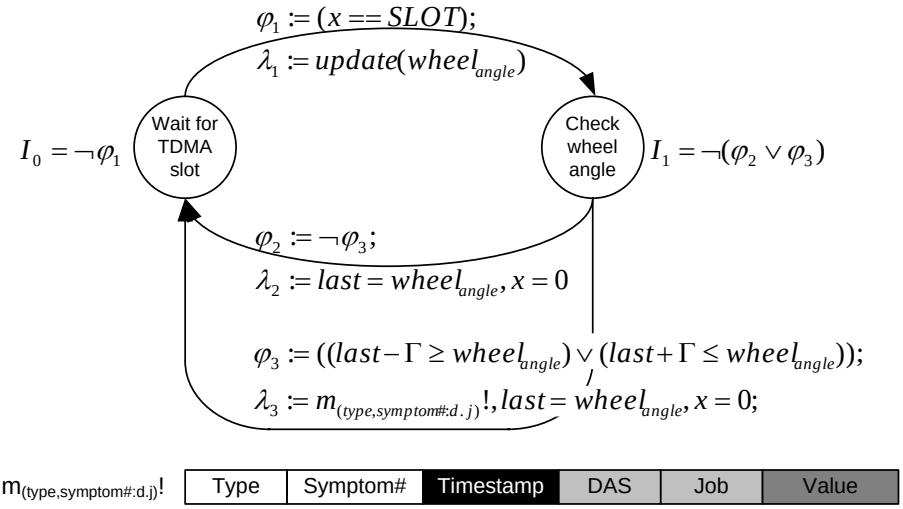

Figure 6.14: Timed Automaton for Symptom: *Out-of-Norm Steering Wheel Angle*

steering wheel angle can change between tow consecutive updates according to the a priori defined TDMA schedule of the time-triggered virtual network.

Event-Triggered Job

In contrast to time-triggered jobs that act according to state semantics, event-triggered jobs only update the values of the interface state variables in case of value changes. In our prototype implementation of the DECOS architecture, the comfort DAS consists of a number of event-triggered jobs providing comfort functionality to the passengers of a car. For example, "Job0" of the comfort DAS regulates the air condition system. Whenever a change in temperature in the passenger compartment is measured, a message is sent on the virtual event-triggered network of the comfort DAS. Since the temperature is also relevant for jobs of other DASs the value is also exported by means of a virtual gateway. Besides distributing the temperature changes the job processes the driver's steering wheel button commands with respect to the air condition system.

According to the event-triggered symptom detector introduced in Section 5.7 the main difference between monitoring of time-triggered and event-triggered jobs is the fact that progress of the timed automaton is not solely controlled by progression of real-time but depends on the availability of messages sent (in the respective queues) by the event-triggered job under investigation. Such a typical symptom detector for an event-triggered job in pseudo-code notation looks as follows:

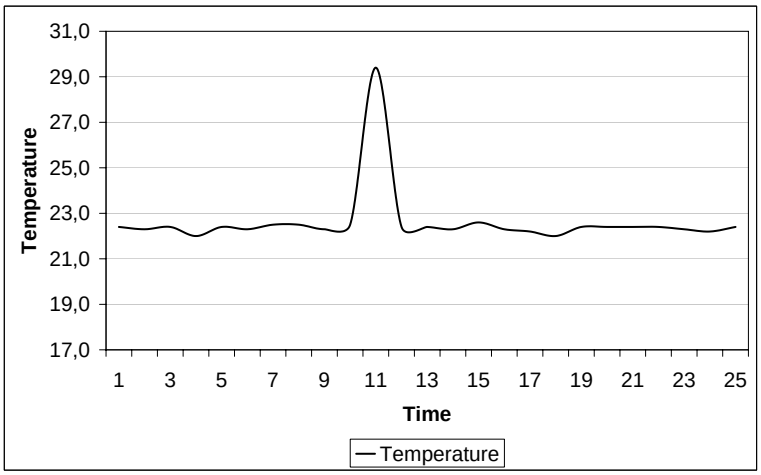

Figure 6.15: Out-of-Norm Measurement of a Temperature Sensor

```
void job0_comfort_symptom1(t_port *port, ...., t_diag_msg *msg) {
  ...
  // read port state variables if message available
  while (avail(port)) {
    // read queue
    et_variable = port->buf;
    ...
    // check values of the port state variables
    if (et_variable == ...) {
        ...
    }
    ...
    // Construct diagnostic message if necessary
    if (something_wrong) {
      msg->data[0] = TYPE_APPLICATION;
      msg->data[1] = APPLICATION_SYMPTOM;
      msg->data[2] = GLOBAL_TIME1;
      msg->data[3] = GLOBAL_TIME2;
      msg->data[4] = NETWORK_COMFORT;
      msg->data[5] = JOB0;
      msg->data[6] = et_variable;
    }
  }
}
```

Consider for instance the measurement curve of a temperature sensor (specified to measure the temperature in the range $-30°$ to $+80°$ Celsius) as depicted in Figure 6.15. The context of use of this senor is the passenger area of a car. Typically, the temperature in the car will be between $-10°$ and $50°$ Celsius, depending on

Field Name	Semantics
Message Type	This field is set to `TYPE_APPLICATION`.
Symptom ID	The unique symptom ID is set to `APPSYM_SENSOR_RAPID_CHANGE` or `APPSYM_SENSOR_AT_THRESHOLD`.
Time Domain	This field includes timestamp of the global time base.
Space Domain	The DAS value of set to `NETWORK_COMFORT` and the job indication field to `JOB0`.
Value Domain	Value of the measured temperature.

Table 6.3: Diagnostic Message for a Out-of-Norm Temperature Measurement

the season of the year. Though the measurement value at $t = 11$ lies within the specified range, the peak represents an anomaly, since it is unlikely that the temperature in the passenger area increases and decreases about 7 degrees Celsius in 2 seconds. Consequently, two application-specific symptom detectors for monitoring

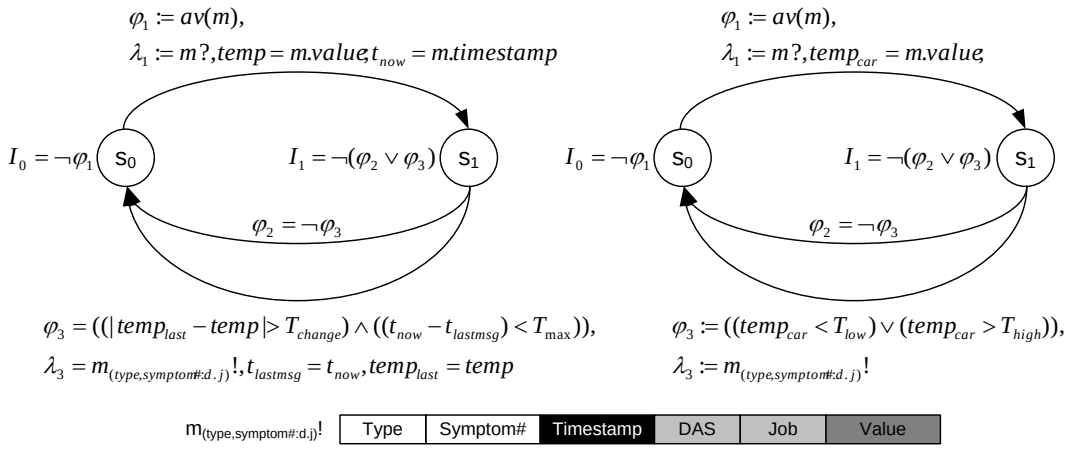

Figure 6.16: Job-inherent Symptom Detection

the port state variables of the event-triggered job controlling the air condition system as depicted in Figure 6.16 are defined. The timed automata on the right hand side implements an algorithm for the evaluating whether the measured car temperature is in the range of the sensor specification (i.e. the limits of the measurement curve). On the left hand of Figure 6.16 a timed automaton is shown, that compares the difference of two consecutive measurements against the threshold value T_{change}. In case the difference exceeds the threshold value and the time difference between the two measurements is smaller than $T_{\max}$, an unlikely rapid change in the temperature value has been detected. For both checks a diagnostic message as specified in Table 6.3 is used.

6.3.2 Job Borderline Faults

In general, the underlying communication model in case of event-triggered jobs is a probabilistic one and specified by means of message service and interarrival

times [Kle75]. This imprecise temporal specification makes error detection in event-triggered communication tricky. In the DECOS diagnostic architecture we provide a solution to error detection for event-triggered jobs by exploiting the global time base of the underlying synchronous core system in order to judge about the compliance of jobs with respect to their interface specification of the event-triggered virtual network.

Since control on the sending and receiving instants is under the sphere of control of the jobs and not the communication system, the diagnostic service monitoring the virtual network service provides detection mechanisms to be parameterized according to the communication model behind the application. This allows not only to detect deviations from the specification, but also to gather facts whether the underlying assumptions behind the communication model hold in reality (i.e. engineering feedback). For instance, unanticipated customer behavior may impose serious problems to the functionality of the system.

Extending the notion of behavior from [GIJ+02], we denote a job's behavior as the sequence of send and receive operations. The behavior of a job is denoted as correct, if it is in accordance with the jobs interface specification. Otherwise we speak of a job failure (i.e. a behavior violating the interface specification) and denote the respective job as faulty. In case of imprecise temporal interface specifications, however, such a demarcation between correct and faulty behavior proves to be difficult. While in a time-triggered communication system the precise temporal interface specification with a priori knowledge about the global points in time of message exchanges allows to definitely distinguish between correct and faulty temporal behavior, the imprecise interface specification of an event-triggered system complicates failure detection. When the temporal behavior of a job is determined by its inputs from the environment, an underlying model of the environment is necessary to evaluate correct job behaviors. For example, consider a user interface job in an automotive application that sends a message to a window lifter job whenever certain buttons are pressed. The ability to detect a faulty user interface job requires assumptions about the frequency and timing for pressing the button. Repeatedly sent messages within a short interval of time might represent a failure of the user interface job, or simply constitute an unanticipated customer behavior (e.g., playing with the window lifter button). Although an omniscient observer can always demarcate between correct and faulty behavior, from within the system the available redundancy and a priori knowledge constrain the ability for performing a definitive classification. In order to handle imprecise temporal interface specifications with limited a priori knowledge, we introduced the concept of an ONA in Section 5.5. A job's behavior is denoted as out-of-norm, if it cannot be classified definitively as correct or faulty at the point in time of occurrence. Out-of-norm behavior represents an improbable behavior that probabilistically represents a failure.

Detection of Bursts. A burst in case of event-triggered communication is the sending of a high number of messages in a short interval of time. A high frequency of bursts can be an indicator for misleading assumptions behind the communication

model or possible sensor failures (e.g., debouncing problems at a button). Whenever a new message is available at the respective port, the timestamp of this message is used by the symptom detector to determine whether the last n (e.g., $n = 5$) messages have been received in the interval $[t1, t2 = t1 + \Delta]$. If this is the case an error message is forwarded and the data structures set to the initial values. Otherwise, the oldest message is discarded, the data structure storing the last $n-1$ timestamps is updated.

Detection of a Violation of the Interarrival Times Specification. The minimum interarrival time between two messages specifies the amount of time that has to elapse between two receive operations messages. By defining and complying with the minimum interarrival times a possible bus overload situation or starvation of jobs can be avoided. Figure 6.17 depicts the automaton implementing this symptom detector.

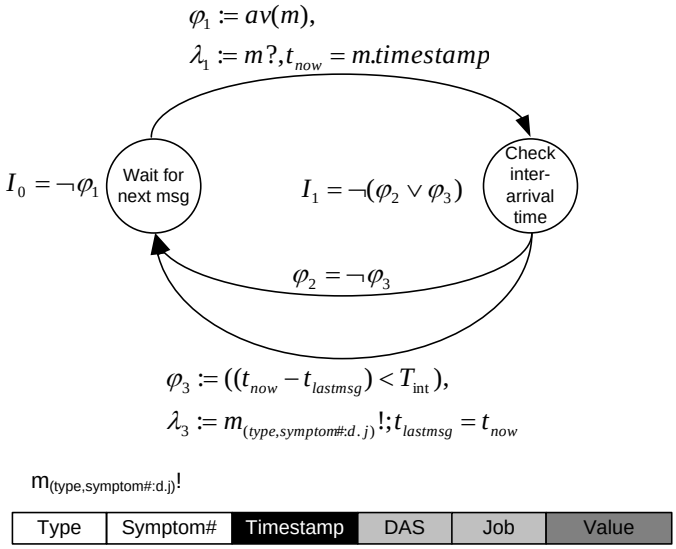

Figure 6.17: Job Borderline Faults: Symptom Detection in case of a *Violation of the Interarrival Time Specification*

In case a new message is available ($av(m)$), the difference in the timestamps of the last message and this new message calculated. In case this value is larger than the a priori specified threshold T_{int} (e.g., 200 ms), a corresponding diagnostic message is generated and the timestamp of the last received updated ($t_{\text{lastmsg}} = t_{\text{now}}$).

Detection of Queue Overflows. The queue status at the input ports of all event-triggered jobs need to be monitored to detect queue overflows at the virtual network service. The virtual network service continuously updates the queue status of the ports of all jobs according to the issued send and receive operations. In contrast to the other symptom detectors for job borderline faults, this job is not triggered by the send and receive operations of a job, but is periodically executed each TDMA slot in accordance with the virtual network service. The assertion simply checks whether a queue overflow has occurred and forwards this information to the diagnostic DAS.

		Code			
Frame Status	**Description**	3	2	1	0
Null Frame	No traffic on the channel	0	0	0	0
Invalid Frame	Coding error, wrong length	0	0	0	1
Incorrect Frame	CRC failed, C-state disagreement	0	0	1	0
Other Error	Mode change permission violated	0	0	1	1
Tentative Frame	C-state/CRC without own membership OK	0	1	0	0
Correct Frame	C-state/CRC with own membership OK	1	0	0	0

Figure 6.18: TTP/C Message Status Field

6.3.3 Component Borderline Faults

The electrical interconnection system typically consists of wiring, connectors, relays, circuit breakers, power distribution panels, and generators. Wiring plays a central role in any distributed system environment, since it provides the infrastructure for exchanging the data between components. Like any other part of the system, the electrical interconnection system is exposed to environmental stress as well as assembly and design faults. Considering that a typical middle class car has about 40 ECUs and approximately 800 wires [POT+05], the likelihood of connector problems is very realistic. In fact, recent studies [SM99, SMM00] indicate that 30% of electrical failures can be attributed to connector problems. Wiring problems, especially connector problems, are difficult to detect, since the inspection itself can be the corrective action (e.g., loose contacts). For this reason, it is important to provide means for the detection of connector failures.

As depicted in Figure 6.18 the TTP/C controller determines the frame status for each received frame. Depending on this message status field, the application can read or discard the received message. This so-called *Error Indication Field* allows an analysis of the status of the communication channels. The TTP/C controller compares the frames from both channels and declares the received frame as correct as long as one frame is correct. This strategy is suitable for application transparent fault-tolerance, however, an integrated diagnostic solution must take the information from both channels into account to enable an investigation of possible physical faults of the replicated channels or bus drivers. In TTP/C we can use the frame status information as described in the following for the detection of borderline failures.

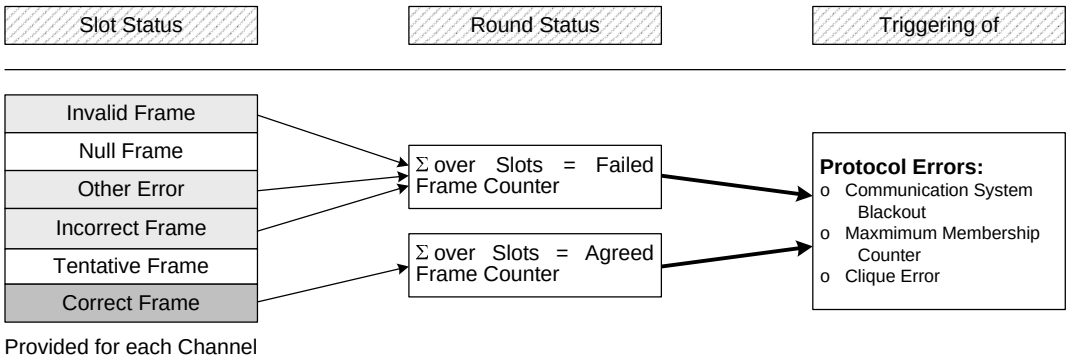

Figure 6.19: TTP/C Protocol Errors

Parameter	**Invalid Frame**
Description	A frame that is syntactical invalid, i.e. coding rules (e.g., coding or expected length) are violated.
Indicator	An invalid frame indicates either a faulty receiver, a faulty sender or a cabling defect. Additional information of other nodes is needed to decide on the origin. For example, in case all other nodes of the cluster perceive the frame as correct, then the receiver or the cabling/connector is faulty. In contrast to a null frame, a signal on the bus can be received (i.e. there are signal edges on the bus that cannot be decoded).
Sampling	Every TDMA slot in case N- or X-frames are received. The message status field of both channels needs to be analyzed for more accurate diagnostic evaluation.

Parameter	**Null Frame**
Description	No activity is observed on any of the channels during a node slot.
Indicator	A null frame indicates either a faulty receiver, a faulty sender (or no sender at all) or a cabling defect. Additional information of other nodes is needed to decide on the origin. For example, in case all other nodes of the cluster perceive the frame as correct, then the receiver or the cabling/connector is faulty. In contrast to an invalid frame, no signal on the bus can be received.
Sampling	Every TDMA slot. The message status field of both channels needs to be analyzed for more accurate diagnostic evaluation.

Parameter	Incorrect Frame
Description	A syntactically valid frame (coding and size correct) for which all CRC checks have failed at the receiver.
Indicator	An incorrect frame indicates rather a transmission fault than an systematic connector fault. For this reason an incorrect frame indicates with higher probability a faulty encoding or decoding unit or an erroneous C-state of the sender/receiver (e.g., hardware fault of the CRC unit).
Sampling	Every TDMA slot. The message status field of both channels needs to be analyzed for more accurate diagnostic evaluation.

Parameter	Tentative Frame
Description	A frame that is correct after the second CRC check. This means the first successor during the acknowledgment considers the controller faulty.
Indicator	Indicates a possible incoming link failure.
Sampling	Every TDMA slot. The message status field of both channels needs to be analyzed for more accurate diagnostic evaluation.

Parameter	Correct Frame
Description	A valid frame which passed the CRC check and all additional semantic checks at the receiver.
Indicator	No diagnostic information.
Sampling	Every TDMA slot.

Besides the frame status field, the CNI of the C2 controller also provides controller registers indicating the relationship of the counter registers with respect to the TTP/C protocol errors as illustrated in Figure 6.19. This includes the *agreed slots counter*, the *failed slots counter*, the *failed frame counter* for channel 0 and 1, and the *null frame counter* for channel 0 and 1 that are described in the following:

- **Agreed Slots Counter.** A counter that counts in each TDMA round the number of nodes that have sent at least one correct frame. It is reset in the own slot.
- **Failed Slots Counter.** A counter that counts in each TDMA round the number of nodes sending at least one failed frame but no correct frame. This counter is reset in the nodes own slot, after the clique avoidance has been

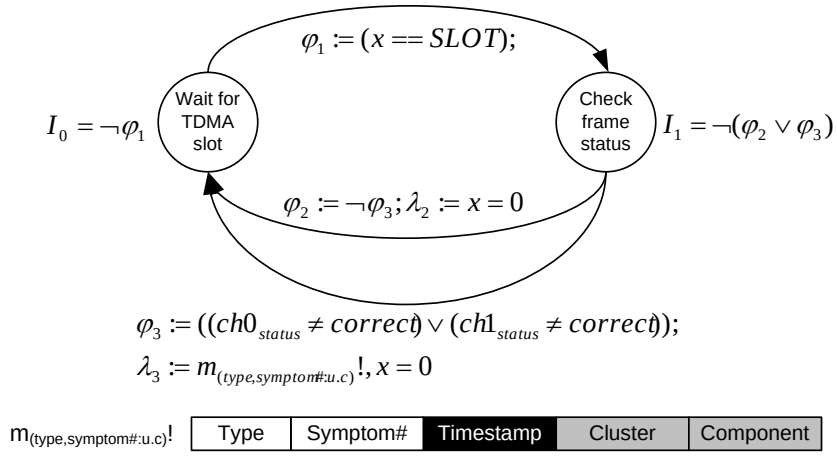

Figure 6.20: Symptom for Component Borderline Faults

performed.

- **Failed Frame Counter Ch0/Ch1.** Number of failed frames (invalid or incorrect frame) on channel 0/1 during the last TDMA round. The counter is reset at the start of the controller's own node slot (like the agreed slots counter).
- **Null Frame Counter Ch0/Ch1.** Number of empty slots (i.e. no activity is observed on any of the channels during a node slot) on channel 0/1 during the last TDMA round. The counter is reset at the start of controller's own node slot.

In Figure 6.20 a timed automaton is depicted that performs a check on the frame status of each physical channel. The timed automaton performs the check on the status of the network in each TDMA slot (using clock variable c) according to the cluster schedule of the core network. Whenever, a new TTP frame is received and the `exchange` data structure updated accordingly, the execution of the timed automaton progresses and the frame status is evaluated. In case the frame status of channel 0 or 1 is other than *correct*, a corresponding diagnostic message as described in Table 6.4 is constructed and forwarded to the virtual diagnostic network. This send operation is specified using $m_{(\text{type,symptom\#},u.c:)}!$ in the automaton.

By replacing the generic check φ_3 stated in Figure 6.20 with a macro providing access to the respective registers of the physical communication controller the timed automaton can be executed on the target platform. In our prototype implementation this is realized by accessing the corresponding values in the `exchange` data structure using

```
frame_status_channel0 = (exchange->slot[index].ch0[1] & 0x000F);
frame_status_channel1 = (exchange->slot[index].ch1[1] & 0x000F);
```

All other C2 controller registers can easily be accessed according to the register layout of the C2 controller [TTT02b], since there exists a one-to-one mapping between the

Field Name	Semantics
Message Type	This field is set to `TYPE_SYSTEMIC` in order to differentiate the message from job-specific symptoms.
Symptom ID	The unique symptom ID is set to `SYM_CORRECT_CH[0/1]`, `SYM_NULLFRAME_CH[0/1]`, `SYM_INVALID_CH[0/1]`, `SYM_INCORRECT_CH[0/1]`, `SYM_OTHER_CH[0/1]`, `SYM_TENTATIVE_CH[0/1]` or `SYM_UNKNOWN_CH[0/1]`.
Time Domain	This field includes timestamp of the global time base.
Space Domain	This field contains the identification of the physical cluster and component.

Table 6.4: Diagnostic Message for for a Component Borderline Symptom

memory-mapped I/O addresses of the C2 controller and the layout of the data in the `exchange` data structure.

6.3.4 Component Internal Faults

Similar to the previous example for the detection of component borderline failures, we present a timed automaton that monitors a component in order to detect component internal faults. In time-triggered communication systems the correct functionality of the time service of each component is of paramount importance for the functionality of the cluster. In the timed automaton as shown in Figure 6.21 we monitor the value of the clock state correction term as our symptom in order to reason about the health status of the quartz. In case the correction term of the clock synchronization algorithm is close to $\Pi/2$, where Π denotes the precision, the symptom fires and a corresponding diagnostic message is forwarded to the diagnostic subsystem. This way unexpected quartz drifts (i.e. due to wearout effects or rapid temperature change [Sch96]) can be detected at an early stage. Note that although this check represents an internal one and is not based on the cross-checking principle, a node sending such information is still operating according to its specification. Consequently, this information supports condition-based maintenance for DECOS components by providing diagnostic information about possible future component failures. The TTP C2 controller provides the following controller registers that can be used for accurate diagnosis of the core communication system. In addition to the clock state correction term field, the measured time difference field provides important information on the arrival time of the frame within the receive window.

Parameter	Clock State Correction Term
Description	This signed term is calculated by the clock synchronization algorithm and given in units of C2 controller microticks. If the absolute value of the total correction term is larger than $\frac{\Pi}{2}$, the node raises a synchronization error.
Indicator	In combination with the *Measured Time Difference Ch0/Ch1* field, this parameter contains information regarding the quality and operation of the quartz. In case this value is larger than $\frac{\Pi}{2}$, the node raises a synchronization error. Therefore, a continuous evaluation of the correction term can detect spontaneous deviations from normal behavior due to transient faults (e.g., EMI) or environmental stress conditions (e.g., temperature). Furthermore, steadily increasing drift rates can be used to predict wearout and subsequent failure of the timing subsystem of the node.
Sampling	Every TDMA slot (in case the CS flag is set to 1).

Parameter	Measured Time Difference Ch0/Ch1
Description	This field holds the arrival time of the frame within the receive window, i.e. the time difference between the expected arrival time of the frame and the real arrival time of the frame on channel 0/1. A negative value states that the actual receipt of the corresponding frame occurred earlier than expected. This indicates that the receiving controller's clock is running lower than the sender's clock (a positive value indicates the opposite). This field is only valid in case the received frame was valid and the frames from the sender are used for clock synchronization (i.e. the SYF flag is set for this slot).
Indicator	By continuously monitoring of this information a systematic timing problem of the node can be identified in contrast to transient disturbances. In case the drift rate increases over time, this information can be used to predict future timing failures resulting from wearout of the quartz.
Sampling	Every TDMA slot in case the SYF flag is set.

6.3.5 Component External Faults

As discussed in Section 3.5, a suitable indicator for wearout of electronic devices is the increase of transient failures in the system. Based on this reasoning, we exploit

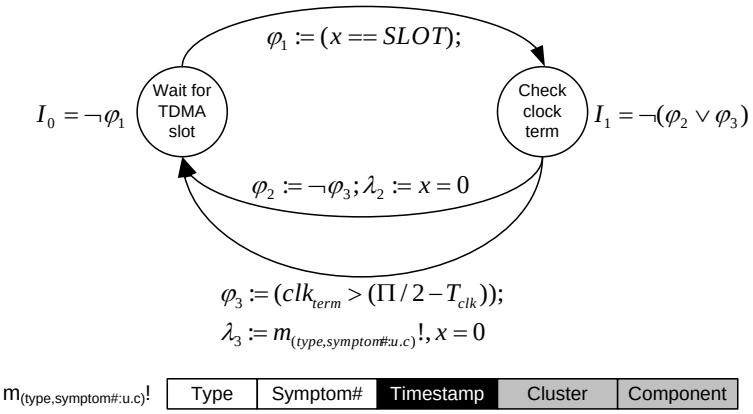

Figure 6.21: Symptom for Component Internal Faults

the membership service of TTP [BP00, KGR91] as in indicator for both transient internal and transient external faults. The question whether the component was affected by a transient fault source or an internal one can only be answered by continuously monitor the operational state of the physical component. According to the results presented in [Con02], transient internal failures tend to occur at a higher frequency and affect the same physical entity. By taking these facts into account, the membership service provides a consistent system wide view on the operational state of the components in the cluster. In addition, by analyzing the restart rate of the components, important statistical data can be collected that serves as engineering feedback also for the design of fault-tolerance mechanisms and strategies. This way the transient failure rate of electronic devices in the field can be determined and the resources optimized accordingly.

In the TTP C2 controller CNI the following status registers are provided for reading membership information. For each component of the distributed system a bit in the membership vector is reserved and either set to 1 when the node operates according to its specification, or 0 in case the majority of the nodes in the cluster consistently declare the node as faulty. Note, that typically before the detection of a membership loss, a diagnostic message indicating a null frame detection is sent.

Parameter	**Membership Flags (Vector)**
Description	A vector that has a unique flag assigned to each member node. If this flag is set, the member node was operational at its last membership recognition point, otherwise it was not operational.
Indicator	The membership provides a consistent view on the operational state of the nodes of the system.
Sampling	Every TDMA slot.

Parameter	Membership Failure Counter
Description	This counter is used to count the number of successive membership failures.
Indicator	The counter is incremented each TDMA round in case it has a cleared membership bit. If the node remains non operational for *Maximum Membership Failure Count* number of TDMA rounds, the controller switches into freeze. In case the maximum counter is set to 0, the number of successive membership failures is not bounded. When the node is operational again (the membership is set to 1), the membership failure counter is not decremented, but reset to 0.
Sampling	Every TDMA round.

6.4 Implementation of the Analysis Algorithms

In the following we describe the realization of the diagnostic DAS in the prototype setup. For evaluating the introduced concepts we decided to implement a centralized analysis DAS with only one job that is considered not to be subject to any faults. However, the design of the analysis data structures and algorithms can easily be extended to a distributed solution. At first, we discuss the basic execution scheme of the analysis job as elaborated on in Section 5.7 and the implemented data structures for keeping track of the experienced failures or anomalies (for both, functional and physical entities of the cluster). The data structures are used to model the system structure according to the maintenance-oriented fault model and thus, allow storing valuable statistical information for both, fault analysis for maintenance purposes, and engineering feedback for the continuous improvement of the system design. By storing the information of the data structures in persistent memory, the analysis algorithm can operate using history information on the health state of the cluster in every new cycle of operation. On the basis of exemplary analysis algorithms we demonstrate how the specified timed automata for analysis are implemented and executed by the analysis job.

6.4.1 Design of the Analysis Job

As depicted in Figure 6.22 the schedule for the component hosting the analysis job is different from the other components in the cluster, since the analysis job is the only non safety-critical application that is executed by the component. Figure 6.23 shows this schedule. Within the 10 ms TDMA round the analysis job is scheduled three times, one slot is reserved for the execution of the high-level services, and one for the execution of standard Linux as a background task.

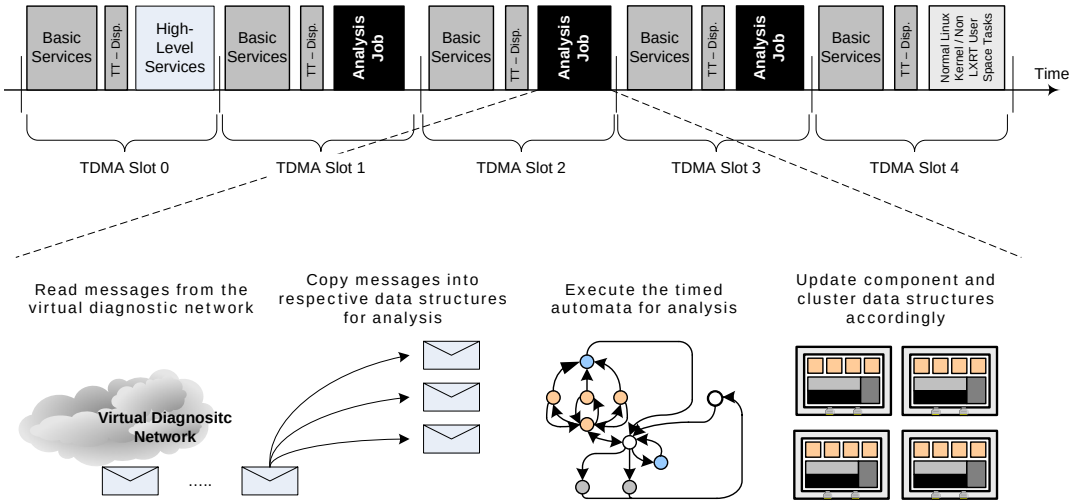

Figure 6.22: The Analysis Job

The LXRT task implementing the analysis job can be decomposed into four subtasks that are executed by every task activation of the time-triggered dispatcher.

1. **Read messages from virtual diagnostic network.** All incoming diagnostic messages – on all three channels – are pulled from the incoming ports of the links of the analysis job.

2. **Copy messages into corresponding ONA analysis data structure.** Depending on the header information of the diagnostic message, a copy of the message is pushed into the queue of each ONA analysis automaton that operates on this information. Such a copy operation is necessary for implementing the cross-checking principle. Note, that for every specified ONA there is a dedicated queue for buffering all incoming systemic or application-specific symptoms that are associated with this ONA. By calling the `av(m)` function as part of the specification of the timed analysis automaton, the availability of a message can be tested (`av(m)` returns the type of the symptom). This message is then finally pulled out of the queue and processed by the automaton whenever the action `m?` is executed.

   Furthermore, the 16 bit timestamp that is included in every diagnostic message derived from the TTP/C global time is extended to 32 bit, since for some analysis algorithms, the timing resolution of 16 bit is not sufficient. For example, in case of a 10 ms TDMA round schedule and a macrotick of 5μsec, the horizon is approximately 330 ms or 33 TDMA rounds. By extending this time field to 32 bit, with the same configuration, the horizon is 358 minutes or nearly 6 hours. The 32 bit time field can be extended to the 64 bit time standard defined by the OMG [OMG03], however, such an extension would require additional bandwidth on the time-triggered core network, while a synchronized 32 bit node local time is supported by the used TTP hardware.

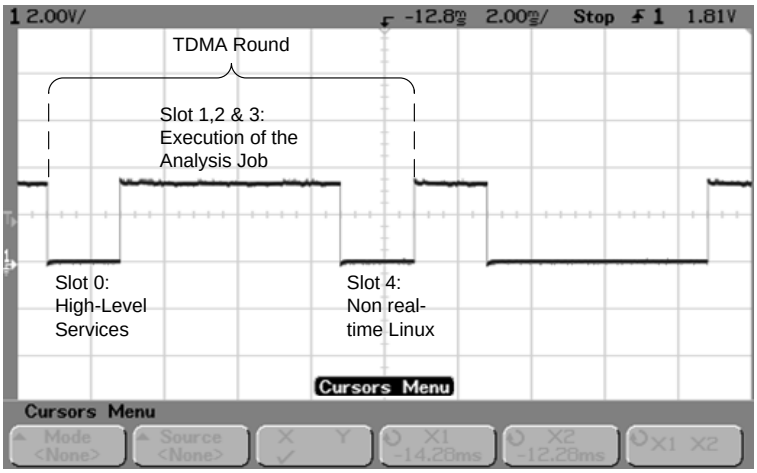

Figure 6.23: The Schedule of the Analysis Job

3. **Execution of the timed analysis automata.** The analysis job executes the timed automata for the ONA analysis according to the execution model presented in Section 5.7.4. By sequentially calling the respective functions implementing the analysis algorithms like in the following code snippet, each timed automaton will be executed as long as a transition is enabled or the simulation time equals the actual global time:

```
analysis_ONA_COMP_0_BORDERLINE_CH0_TA();
analysis_ONA_COMP_0_BORDERLINE_CH1_TA();
...
analysis_ONA_JOB_5_BYWIRE_TA();
...
```

4. **Update of data structures.** As a result of the execution of the timed automata, the data structures containing the health status information of the functional (i.e. DASs and jobs) and physical entities (i.e. components and clusters) are updated. In addition to determining the trust level of each entity of the system, statistics for engineering feedback are also stored and processed.

6.4.2 Analysis Data Structures

Both, Figure 6.24 and Figure 6.25 are designed according to the maintenance-oriented fault model. All ONA analysis algorithms have private internal data structures, however, the result is stored in the global analysis data structure in order to allow:

- **Hierarchical ONA analysis.** As introduced in Section 5.5 the construction of hierarchical ONAs is crucial for a reuse of already thoroughly tested and verified

checks in order to efficiently make use of available resources. Furthermore, by defining an ONA that operates on the results of other ONAs, results of multiple different analysis processes can be included into a global assessment process to determine the health state of the system.

- **Exploitation of the knowledge about the functional and physical system structure.** As already stated, integrated architectures are designed to overcome the "1 Function - 1 ECU" limitation of currently deployed federated systems. From a diagnostic point of view this property of integrated architectures is crucial, since the knowledge about the functional (i.e. jobs and respective DASs) and the physical system structure (i.e. components, core network topology) provides the possibility to distinguish between hardware and software faults affecting the cluster. For a detailed discussion of the main idea refer to Section 5.9.3.

- **Reduced representation.** By storing the results of the individual ONAs analysis algorithms in a common data structure, not only the possibility for a hierarchical analysis is provided, but also a reduced representation of the global health state can be derived. This reduced information captures the major results of the analysis up to a specific point in time, i.e. a ground state (i.e. every 1000 TDMA rounds), and is either stored permanently in a persistent memory to cope with possible faults affecting the component hosting the analysis DAS or disseminated in case of distributed analysis solution.

  This reduced representation is also important when the diagnostic DAS is accessed by the service technician. Here, a detailed analysis of the system state is of limited help. By contrast, a *trust level* for every FRU serves as the basis for determining the correct maintenance action.

Component Analysis

As depicted in Figure 6.24 the data structure `TD_decos_component` for each component captures diagnostic information about the hosted jobs and corresponding DASs as well as hardware-specific information of the time-triggered core network. For each hosted job a `TD_job` data structure captures among meta-information (such as the name of the job) the counter values for job-inherent, job-borderline and job-external faults:

```
typedef struct a_job {
    char                job_name[NAME_LENGTH];
    enum vn_paradigm    paradigm;
    // inherent
    unsigned int        inherent_counter;
    ...
    // borderline
    unsigned int        borderline_counter;
```

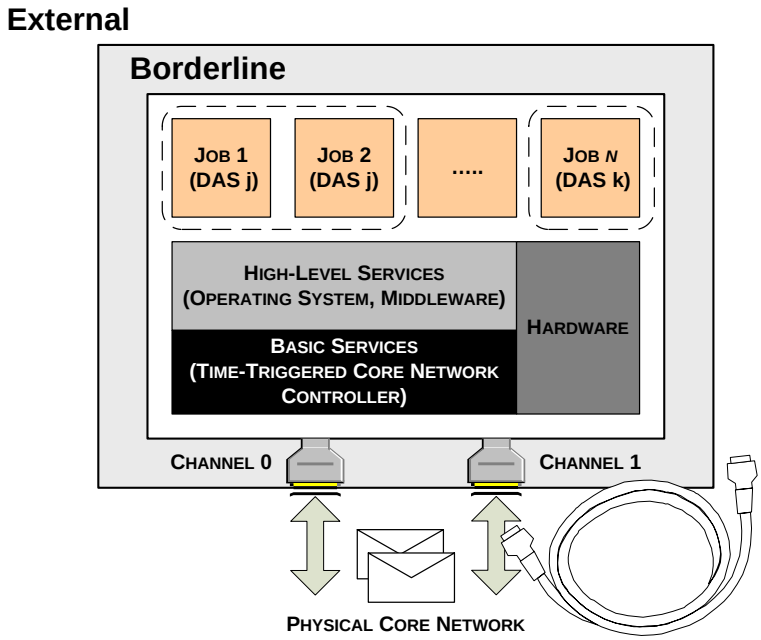

Figure 6.24: Conceptual Model of the Data Structure for Component Analysis

```
    unsigned int        borderline_anomaly_counter;
    unsigned int        send_counter;
    unsigned int        send_anomaly_counter;
    unsigned int        receive_counter;
    unsigned int        receive_anomaly_counter;
    // external
    unsigned int        external_counter;
    ...
} TD_job;
```

In analogy, information about the DAS structure is available for incorporating information about the functional space dimension into the analysis process. Therefore, knowledge about the jobs of the DASs hosted on a component according to the cluster schedule is stored in the `TD_DAS` data structure:

```
typedef struct a_DAS {
    char            DAS_name[NAME_LENGTH];
    unsigned int    number_of_jobs;
    TD_job          job[MAX_NUMBER_JOBS_COMP];
} TD_DAS;
```

Finally, the component data structure `TD_decos_component` combines all diagnostic information to allow a health state assessment according to the analysis concepts introduced in Section 5.9.

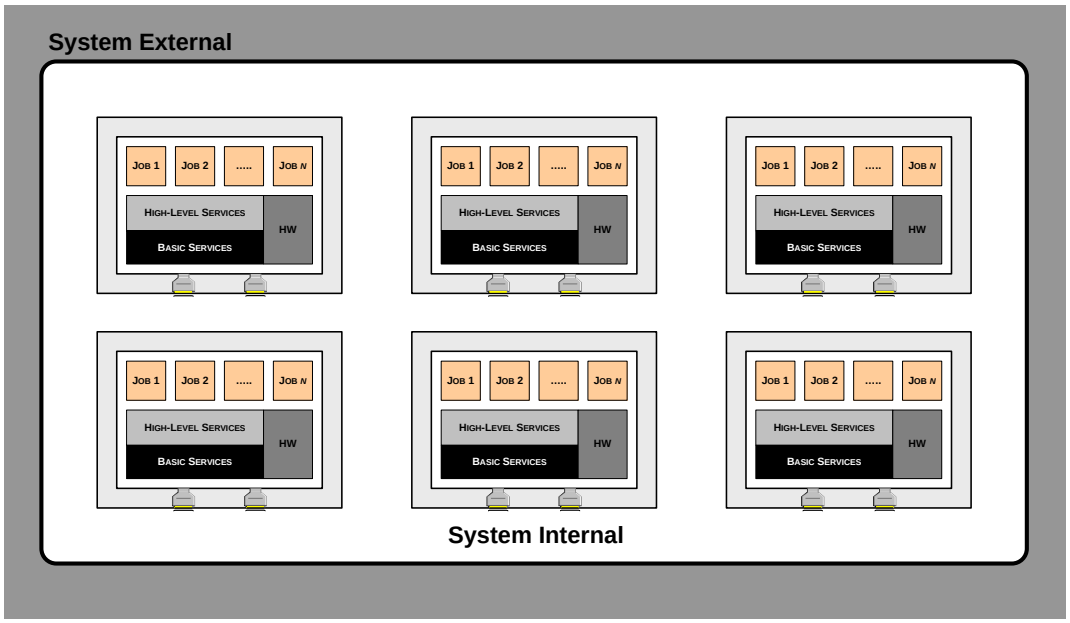

Figure 6.25: Conceptual Model of the Data Structure for Cluster Analysis

```
typedef struct a_component {
    TD_DAS          DAS[NUMBER_OF_DAS];
    unsigned int    number_of_DAS;
    // internal
    unsigned int    internal_counter;
    unsigned int    internal_anomaly_counter;
    ...
    // borderline
    unsigned int    ch0_counter;
    unsigned int    ch0_anomaly_counter;
    unsigned int    ch1_counter;
    unsigned int    ch1_anomaly_counter;
    unsigned int    borderline_counter;
    unsigned int    borderline_anomaly_counter;
    ...
    // external
    unsigned int    external_counter;
    unsigned int    external_anomaly_counter;
    unsigned int    number_of_restarts;
    ...
    // high-level-services
     ...
    // core-services
    ...
} TD_decos_component;
```

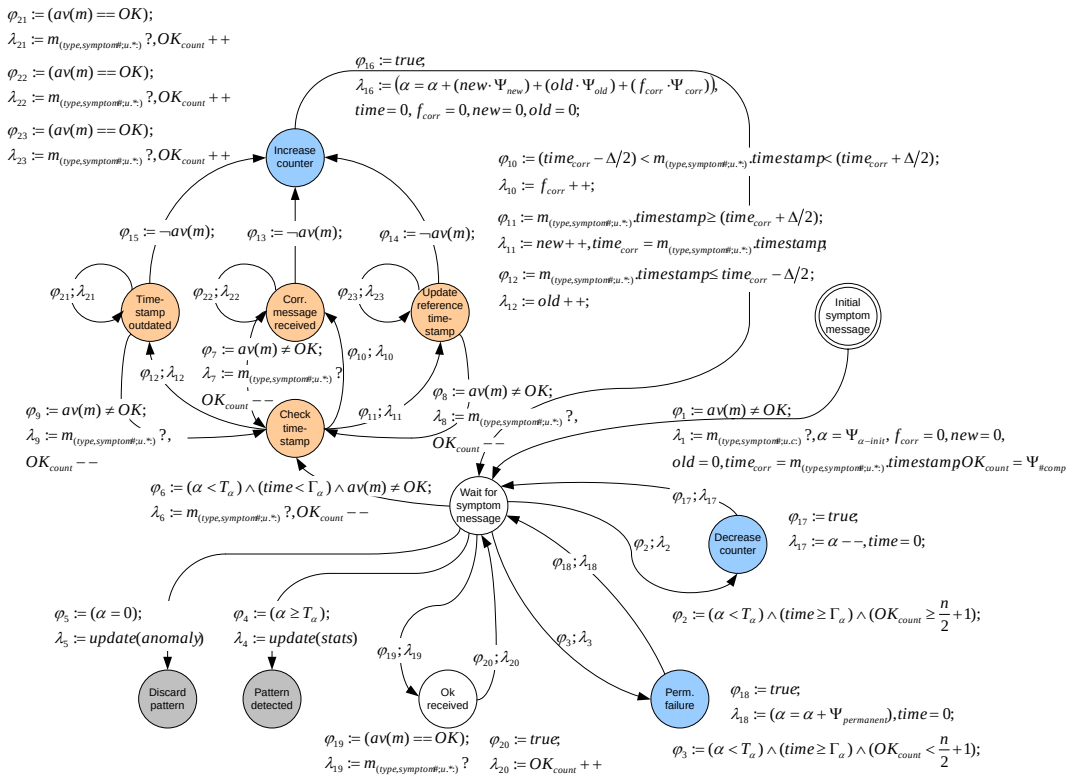

Figure 6.26: Out-of-Norm Analysis: Timed Automaton for the Determination of Component Borderline Faults

Cluster Analysis

In analogy to the component data structure, the cluster data structure `TD_decos_cluster` as shown in Figure 6.25 unites the diagnostic information for each component in the system for a cluster wide assessment. This cluster data structure is crucial for the deployment of hierarchical ONAs as introduced in Section 5.5. For a detailed example see also Section 6.4.5.

6.4.3 Determination of Component Borderline Faults

In the following we discuss the implementation of the analysis algorithms for determining component borderline faults based on the symptoms introduced in Section 6.3.3. For showing the feasibility of the diagnostic framework we employ a simple threshold based analysis scheme as introduced in Section 3.7. However, in contrast to typical analysis algorithms processing only node local information, the timed analysis automaton as shown in Figure 6.26 for the systemic diagnosis of borderline failures correlates information from other components of the cluster to indisputably judge whether a failure has occurred. In case no correlated symptoms have been detected the increase of the α-counter value is marginal. By contrast,

once a correlated information is detected, f_{corr} is increased and the α-counter updated accordingly. To emphasize the importance of such correlated information, the weighting factor Ψ_{corr} is multiplied with f_{corr} and then added to the α-counter value.

Since in our prototype implementation a core communication network with two replicated channels is used, for each component two ONAs for the determination of borderline faults are required (i.e. one for each channel).

Symptom Forwarding and Message Decoding

Once the symptoms that are associated with the ONA are received on the virtual diagnostic network, the symptoms are processed and forwarded to the respective ONA queue. This is a prerequisite for executing the timed automaton that implements the analysis algorithm. The following code snippet shows how symptoms for identifying component borderline failures are processed. Note, that the message field `dmsg.data[6]` contains the identification of the component that is assumed to be faulty by the node that sent the message.

```
while (rcvEtMessage(&(Link2ALD->port[i]), &dmsg)==ET_OK){
  if (dmsg.data[0] == TYPE_SYM) {
    switch(dmsg.data[1]) {
    case SYM_NULLFRAME_CH0:
        // Which component is faulty? -> read value field
        switch(dmsg.data[6]) {
            case 0: target_queue = ONA_COMP_0_BORDERLINE_CH0; break;
            case 1: target_queue = ONA_COMP_1_BORDERLINE_CH0; break;
            case 2: target_queue = ONA_COMP_2_BORDERLINE_CH0; break;
            case 3: target_queue = ONA_COMP_3_BORDERLINE_CH0; break;
        }
    case SYM_NULLFRAME_CH1:
        // Which component is faulty? -> read value field
        switch(dmsg.data[6]) {
            case 0: target_queue = ONA_COMP_0_BORDERLINE_CH1; break;
            case 1: target_queue = ONA_COMP_1_BORDERLINE_CH1; break;
            case 2: target_queue = ONA_COMP_2_BORDERLINE_CH1; break;
            case 3: target_queue = ONA_COMP_3_BORDERLINE_CH1; break;
        }
    // put message into respective ONA queue
    ...
    // update meta-information
    ONA[target_queue].wr_pos++;
    if (ONA[target_queue].wr_pos >= DIAG_MSG_PER_ONA) {
        ONA[target_queue].wr_pos = 0;
    }
    ...
```

Definition of Thresholds and Weighting Factors

Every threshold-based analysis technique depends on the setting of the parameters that determine the behavior of the algorithm, in particular the

- penalty weighting factors ($\Psi_{\text{new}}, \Psi_{\text{old}}, \Psi_{\text{corr}}, \Psi_{\text{permanent}}, \Psi_{\alpha-\text{init}}$)
- α-counter threshold (T_α), and
- timing parameters (Γ_α, Δ).

As shown in Figure 6.26 the threshold value T_α determines when a fault pattern is detected and stored in the previously introduced analysis data structures (`update(stats)`). The parameter Γ_α defines the length of the interval between decreasing the α-counter value in case no symptom message is received. The constant Δ is used for specifying the time window during which correlated information with respect to the previously received symptoms is processed. Whenever a symptom with a timestamp is received that strengthens the belief in the correctness of the previously received symptom(s), f_{corr} is increased. f_{corr} is multiplied by the weighting factor Ψ_{corr} before adding to the α-counter value. In case Ψ_{corr} is set to a larger value than Ψ_{new} or Ψ_{old}, correlated information leads to a significant increases of the α-counter value. After processing all incoming symptoms the α-counter value is updated according to the transition enabled by guard φ_{16} and action λ_{16}:

$$\alpha = \alpha + (new \cdot \Psi_{\text{new}}) + (old \cdot \Psi_{\text{old}}) + (f_{\text{corr}} \cdot \Psi_{\text{corr}})$$

where, *new* holds the number of newly received messages (with future timestamp), *old* holds the number of received messages with outdated timestamp (due to possible delays introduced by the event-triggered virtual network service), and f_{corr} the number of correlated messages.

Implementation of the Timed State Machine

The transformation of the automaton described in Figure 6.26 into executable code is straightforward. In the following we discuss relevant implementation details of the implemented state machine as executed by the analysis job.

Global Time. Since the progression of the execution trace of the automaton implementing the analysis algorithm depends on the progression of real-time, all local clock variables are synchronized with the global time provided by the underlying time-triggered core network. As already stated, the 16 bit global time provided by TTP is extended to a node local 32 bit time field to implement timing constraints exceeding 16 bit (or approximately 330ms in the used schedule and TTP cluster configuration). The 32 bit timestamp is derived from the `exchange` data structure (see Section 6.2) as follows

```
#define GLOBAL_TIME32
 (exchange->slot[nComponent].ctrl[14] << 24) + \
 (exchange->slot[nComponent].ctrl[15] << 16) + \
 (exchange->slot[nComponent].status[0] << 8) + \
  exchange->slot[nComponent].status[1])
```

Thereby, the TTP specific time fields are made available for the LXRT analysis job. Of particular interest with respect to the progression of real-time is state `CHECK_TIME_STAMP`, where the following timing analysis is performed:

- In case $\varphi_{10} := (\text{time}_{\text{corr}} - \frac{\Delta}{2}) < m_{(\text{type,symptom\#},u.*:)}.\text{timestamp} < (\text{time}_{\text{corr}} + \frac{\Delta}{2})$ is enabled, a correlated diagnostic message has been detected and the correlation factor f_{corr} is increased. This condition shows how a holistic view on the distributed state of the integrated system can be exploited for a more accurate diagnosis of experienced failures.

- In case $\varphi_{11} := m_{(\text{type,symptom\#},u.*:)}.\text{timestamp} \geq (\text{time}_{\text{corr}} + \frac{\Delta}{2})$ evaluates to $\mathbb{T}$, a message with a new symptom has arrived. As a consequence the *new* counter is increased and the variable holding the reference timestamp $\text{time}_{\text{corr}}$ is updated accordingly.

- In case $\varphi_{12} := m_{(\text{type,symptom\#},u.*:)}.\text{timestamp} \leq (\text{time}_{\text{corr}} - \frac{\Delta}{2})$ a message with an outdated timestamp has arrived. This condition might fire, due to possible message delays at the virtual event-triggered network. If the guard φ_{12} evaluates to $\mathbb{T}$, the *old* counter is increased and later added to the α-counter value.

Although a node local 32 bit timestamp is used in the implementation, a time overflow might occur. This has to be taken into account when transforming the guards into executable code.

Event Semantics. The use of event semantics for encoding diagnostic information has the advantage of effective usage of the available bandwidth for diagnosis. Since only changes in interface state variables are distributed, a failure is active as long as no message is received, stating that a correct state of the variable has been restored. Whenever no confirmation messages arrive, a permanent failure is assumed. This issue has to be taken into account when implementing the analysis algorithm.

Therefore, the variable OK_{count} holds the number of received messages indicating that the status of the channel is correct. If the message data field contains the symptom `SYM_CORRECT_CH0/1` the variable OK_{count} is incremented, otherwise decremented (see also the transitions φ_6, λ_6 and $\varphi_{19}, \lambda_{19}$ in Figure 6.26). In the correct case OK_{count} equals the number of components in the system. Consequently, the function `av(m)` needs to return the symptom value for implementing this strategy. As depicted in the timed automaton, in case the value of OK_{count} is smaller than the number of the deployed physical components, there are two possible transitions:

- If $OK_{\text{count}} \geq \frac{n}{2} + 1$ evaluates to $\mathbb{T}$, then the majority of the nodes in the cluster have confirmed that the channel status is correct again. In case OK_{count} does not equal the number of components, the counter is increased over time. In the implementation every $5 \cdot \Gamma_\alpha$ time units, OK_{count} is incremented by one to compensate for missing "OK messages".
- If $OK_{\text{count}} < \frac{n}{2} + 1$ evaluates to $\mathbb{T}$, then only a minority of the nodes in the cluster have confirmed that the physical connection to the time-triggered core network has been reestablished. In this case a permanent failure has been detected and the α-counter value is increased by $\Psi_{\text{permanent}}$ every Γ_α time units as long as either the missing "OK messages" arrive or the threshold T_α is exceeded.

Exceeding the Threshold T_α. In case the α-counter value exceeds T_α, the global analysis data structure is updated accordingly. For example, in case α-counter for channel 0 of component 2 exceeds T_α, the following code

```
cluster.component[2].ch0_counter++;
cluster.component[2].borderline_counter++;
```

is executed. Then, the timed automaton is reset and the execution starts over again. Reoccurring overstepping of the threshold then results in additional ONA firing and thus strengthen the belief in the previous analysis results. However, alternative strategies can be implemented. For example, after exceeding the threshold T_α the component can be declared as permanent faulty and scheduled for maintenance action.

Decrease of the α-counter. In case $\alpha \leq 0$ the fault pattern for a channel fault is discarded. Since the diagnostic architecture follows the record any single anomaly design principle, the anomaly counters for the respective component are increased:

```
cluster.component[2].ch0_anomaly_counter++;
cluster.component[2].borderline_anomaly_counter++;
```

Subsequently, the timed automaton is reset and the execution starts over again by initializing the automaton data structures.

Parameter Settings

In the implementation of the threshold-based analysis algorithm depicted in Figure 6.26 the following values for the parameters are used:

```
#define INIT_ALPHA_VALUE          20
#define T_ALPHA                  300
```

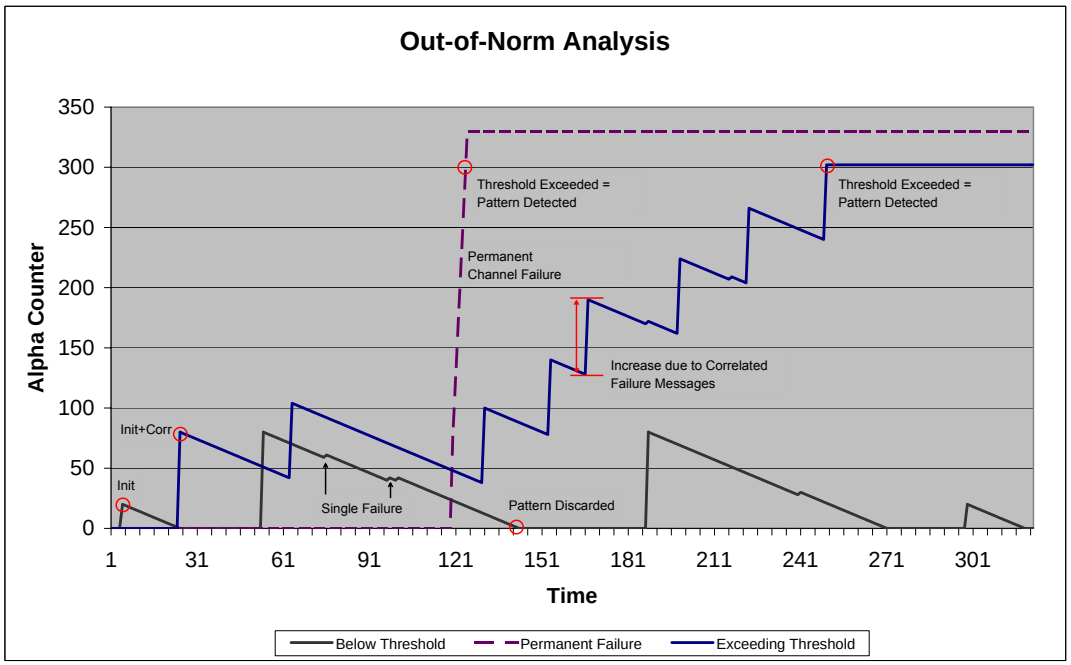

Figure 6.27: Out-of-Norm Analysis: Alpha Counter for Different Faults

```
#define GAMMA_ALPHA          0x5B8D80 // (*5musec = 30 Seconds)
#define DELTA                0x0007D0 // one TDMA round [-3E8,+3E8]
#define PSI_CORR                   30
#define PSI_NEW                     2
#define PSI_OLD                     5
#define PSI_PERMANENT              50
```

`GAMMA_ALPHA` is set to `0x5B8D80` that equals an interval of 30 seconds. In case no further symptom is received within 30 seconds, the α-counter value is decreased by one. Messages with a timestamp differing no more than one TDMA round (or `DELTA` = `0x7D0` macroticks) are assumed to be correlated. This opens the time window `[t-3E8,t+3E8]` for processing correlated information. A single symptom increases the α-counter value only by 2 as defined in `NEW_FACTOR`, whereas any additional correlated symptom message increases the α-counter value by 30 corresponding to `CORR_WEIGHT_FACTOR`. The parameter `INIT_ALPHA_VALUE` defines the penalty for the first occurrence of a symptom, and `PSI_PERMANENT` increases the α-counter value by 50 every `GAMMA_ALPHA` time units in case a permanent channel failure is detected.

Results

Figure 6.27 depicts the measurement results of three experiments analyzing the performance of the implemented α-counter strategy. On the basis of three chosen representative fault types, namely

- Permanent internal channel failure (denoted as "Permanent Failure")
- Transient internal channel failure (denoted as "Exceeding Threshold")
- Transient external channel failure (denoted as "Below Threshold")

we describe the measurement curves of the α-counter values in more detail. The x-axis of the diagram presented in Figure 6.27 shows the progression of time. Each time unit corresponds to Γ_α, i.e. the time that has to elapse without an error messages to decrease the value of the α-counter. The y-axis represents the value of the α-counter.

Interpretation of the Results

Permanent internal channel failure. As depicted, once a permanent channel failure occurs, the value of the α-counter is increased by $\Psi_{\text{permanent}}$ every Γ_α macroticks. Since no message with symptom `SYM_CORRECT_CH0/1` is received, indicating that the physical link is successfully established again, and the majority of the components in the cluster support this view ($OK_{\text{count}} < \frac{n}{2} + 1$), an increase of the α-counter value exceeding T_α is the result. Consequently, the channel is declared as being permanent faulty and a corresponding entry into the diagnostic data structures is written.

Transient internal channel failure. In today's automotive bus systems the majority of recorded problems are due to communication problems on the CAN bus. Consequently, the diagnosis of communication blackouts on the deployed bus systems is of critical importance. Since transient internal channel failures require maintenance action such as inspection of the cable loom or change of a defective ECU, accurate diagnosis is vital to keep warranty costs low. The following points as highlighted in Figure 6.27 are of special interest. At $t = 23$ a channel failure is detected and processed, resulting in an increase of $\Psi_{\alpha-\text{init}}$ of the α-counter value. Since correlated failure messages confirm this channel failure, the term $new \cdot \Psi_{\text{new}} + f_{\text{corr}} \cdot \Psi_{\text{corr}}$ is added to the value of the α-counter. As depicted, the continuous detection of symptoms indicating a potential internal channel failure causes that the fault pattern is never discarded. In fact, the reoccurring correlated symptom detections result in an increase of the α-counter value beyond T_α as time progresses.

Transient external channel failure. According to the maintenance-oriented fault model, transient external faults are falling into the category of faults where no maintenance action is required to restore the intended functionality. Whenever, the first diagnostic message with a symptom that is relevant for the analysis of component borderline faults is processed the α-counter is increased according to $\Psi_{\alpha-\text{init}}$. The curve presented in Figure 6.27 shows this increase in the beginning, due to an unconfirmed symptom message. Whenever the α-counter value, that decreases over time, is 0 again, the fault pattern is discarded and the timed automaton reset after

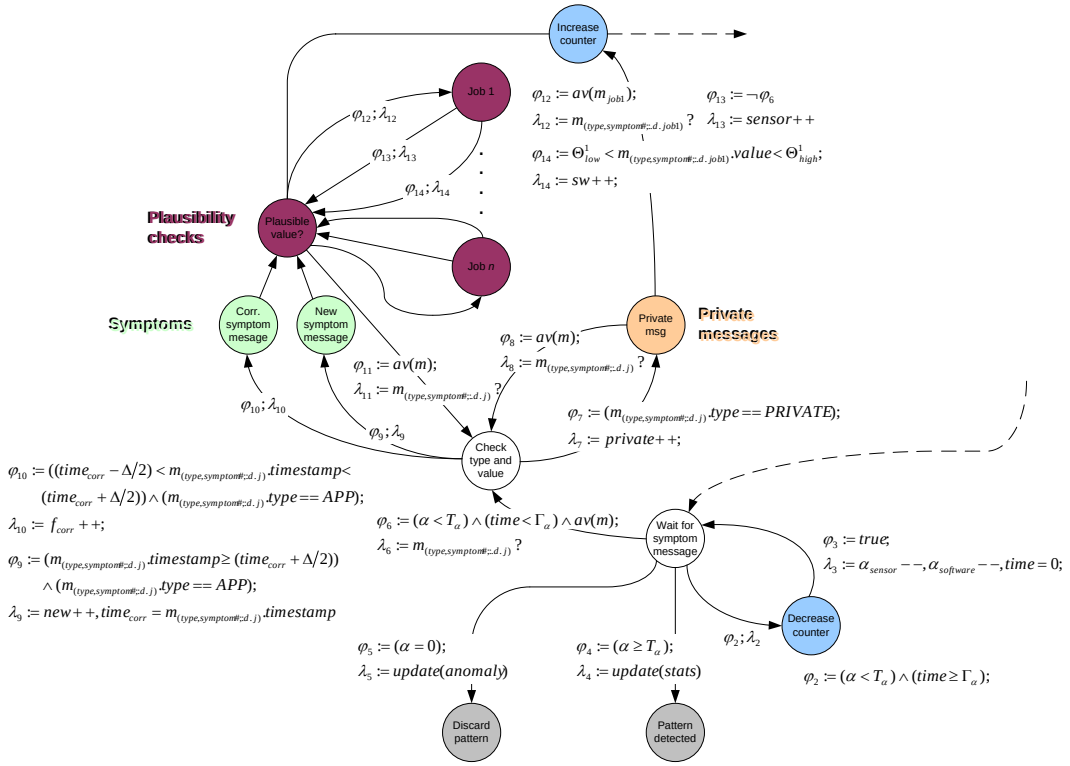

Figure 6.28: Out-of-Norm Analysis: Timed Automaton for the Determination of Job Inherent Faults

writing an entry into the analysis data structures. Then the execution of the timed automaton restarts in state "initial symptom message". The second increase of the α-counter value at time $t = 50$ is much higher, since in this case the channel failure is confirmed by all other nodes on the cluster. In contrast to the single failure messages at time $t = 75$ and $t = 100$, where only Ψ_{new} is added to the α-counter value having only a marginal impact on the outcome of the analysis result. Note, that although the ONA never fires, i.e. the threshold T_α is never exceeded by the α-counter value, every entry made into the analysis data structures whenever the fault pattern is discarded is an indication for an anomaly. This data can provide important feedback when analyzing a large population of maintenance records.

6.4.4 Determination of Job-Inherent Faults

Similar to the previous example, the analysis of job-inherent faults is realized using a threshold-based analysis algorithm. In contrast to systemic diagnosis, however, the diagnosis of job-inherent faults depends on profound knowledge on the application and must be specified by the application designer.

The implemented analysis algorithm for the analysis of job-inherent faults, i.e. these are either faults affecting the controlled sensor or actuator or software

design faults, is depicted in Figure 6.28 and similar to the algorithm presented in Section 6.4.3. For the analysis the following information can be used to improve the accuracy of the analysis process:

- **Symptoms Generated by the Message Classification Layer.** The message classification layer as introduced in Section 5.8 realizes the cross-checking principle at application level. Even executed at the same component, the message classification provides means to independently monitor the output of a job under the DECOS fault hypothesis (cf. Section 4.6) which states that jobs are the FCR with respect to software faults. Similar to Section 6.4.3, correlated symptoms have a significant impact on the α-counter value. The main difference is the fact that for the analysis two α-counter values are used, namely α_{sensor} and α_{software} for determining the most likely fault source.

- **Private Symptom Messages.** Although not designed according to the cross-checking principle, job internal private diagnostic messages encoding symptoms not made public at the interface for IP issues can be effective revealing faults affecting the attached I/O. Since not all control registers of the transducer are mapped into the port state of the job and thus made available for cross-checking, this internal information provides additional information confirming or falsifying a fault analysis hypothesis. Depending on the type of provided information, either α_{sensor} or α_{software} is increased.

- **Plausibility Checks.** Interface state variables of other jobs can be used for a plausibility analysis. For example, interface state variables that are read from the environment, either by a replicated sensor, or a sensor type that allows deducing a wrong/contradicting measurement, can be used for improving the result of the analysis process. This implies that not only diagnostic messages from the virtual diagnostic network but all other messages of the system can be used as input. As a consequence, the analysis job needs ports to the respective virtual networks. In the depicted automaton, after receiving a symptom, the belief in this provided information is strengthened or weakened by a plausibility analysis where possible. Such a typical check is performed by guard φ_{12}. In case of an event-triggered virtual network, a message must be available at the respective input port (in case of a time-triggered virtual network the guard evaluates always to $\mathbb{T}$). Subsequently, the value of the message of the input port of the respective job is evaluated in oder to determine whether the symptom is a strong indication for a software or a transducer fault. Furthermore, the values can be used for gathering statistical data as required for engineering feedback.

In case of a detected or discarded fault pattern the `TD_job` data structure is updated accordingly for a DAS-wide analysis to determine whether the fault/anomaly affects only one job or more jobs at the same time.

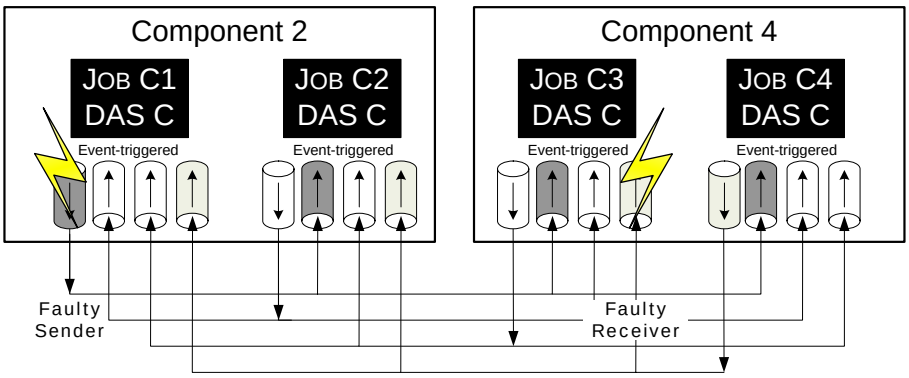

Figure 6.29: Diagnosis of Event-Triggered Virtual Networks: Faulty Sender vs. Faulty Receiver

6.4.5 Determination of Job Borderline Faults

Event-triggered virtual networks are frequently used as the communication infrastructure for non safety-critical jobs as used to provide comfort functionality in automotive systems. As discussed in detail in Section 5.4.2, the impreciseness of the temporal LIF specification due to the underlying probabilistic communication models makes a definite classification of a job difficult. The lack of temporal rigidity in the interface specification makes it hard to judge whether a job conforms to the specification and thus diagnosis of event-triggered systems is prone to misjudgement. Although interarrival and service times can be specified, the limits are not hard and thus a missed deadline is no more than an anomaly and cannot be judged as an indisputable failure. However, the accurate diagnosis of the communication activities of event-triggered jobs is vital to

- identify configuration faults (i.e. analyze whether the underlying communication model holds in reality), and
- to identify reoccurring job failures as a good indication of software faults or a defective sensor/actuator connected to the job.

In the DECOS diagnostic architecture we exploit the architectural services to improve the accuracy of the diagnosis for event-triggered jobs significantly. Although we cannot ignore the impossibility result of [FLP85], by operating on the global state of the system we can include the three dimensions of time, space, and value to judge about the correct functionality of each job:

- **Space Domain.** The inclusion of the space domain into the analysis process allows determining the limits with respect to the functional and physical entities of the fault. In the physical space domain we distinguish between *component-local* or *cluster-wide* (i.e. a fault affecting more than one component). In the functional space domain we distinguish between *job-local* or *DAS-wide* (i.e. a fault affecting more than one job within the respective DAS). See also Figure 5.2.

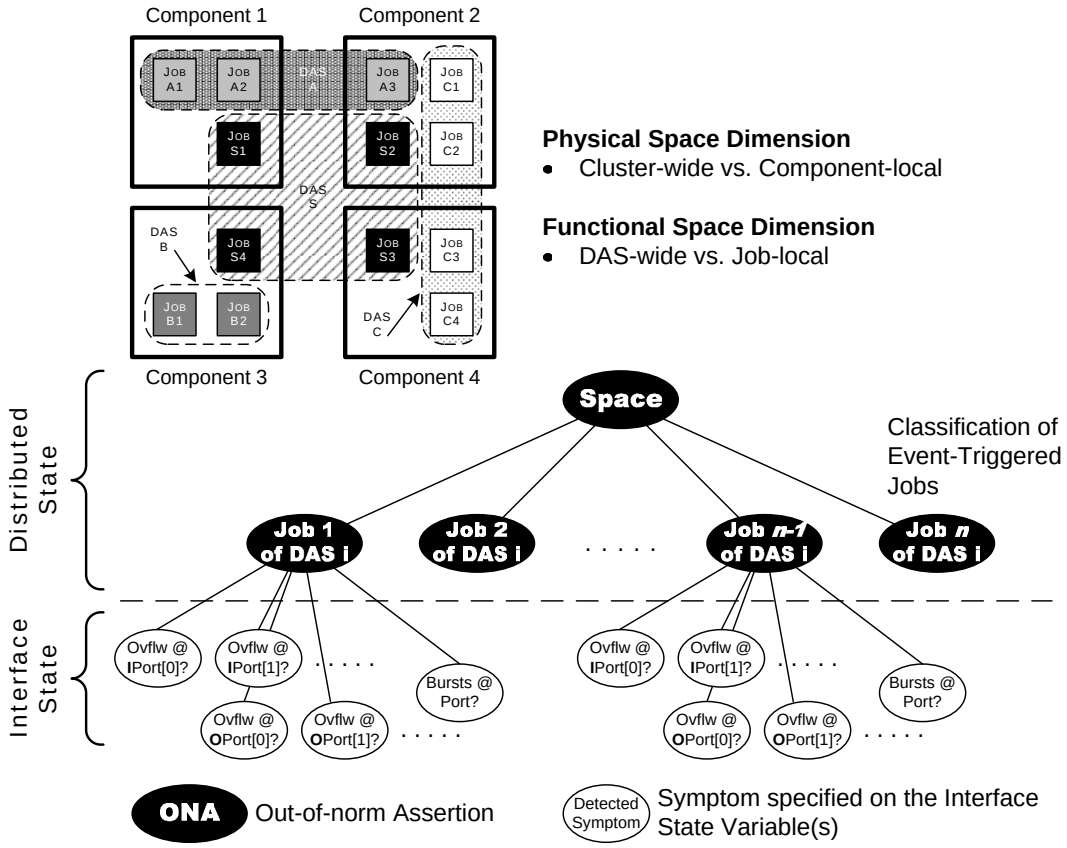

Figure 6.30: Hierarchical Out-of-Norm Assertions

- **Time Domain.** By exploiting the global time for timestamping of symptoms, a possible correlation of diagnostic events can be identified and processed.
- **Value Domain.** Depending on the type of symptom (e.g., queue overflow, single burst) different analysis strategies can be implemented. For example a queue overflow can lead to a more significant increase of the α-counter value than a single burst. Additionally, information provided by the operating system can be used to judge about the computational progress of a job.

In the prototype setup, for evaluating the correctness of either the underlying communication model behind the parametrization of the virtual network service and the correctness of the job software the following symptoms are processed by the analysis algorithm:

- Queue overflows at the input ports of a receiving job
- Queue overflows at the output ports of a sending job
- Burst anomalies at the output ports of a sending job

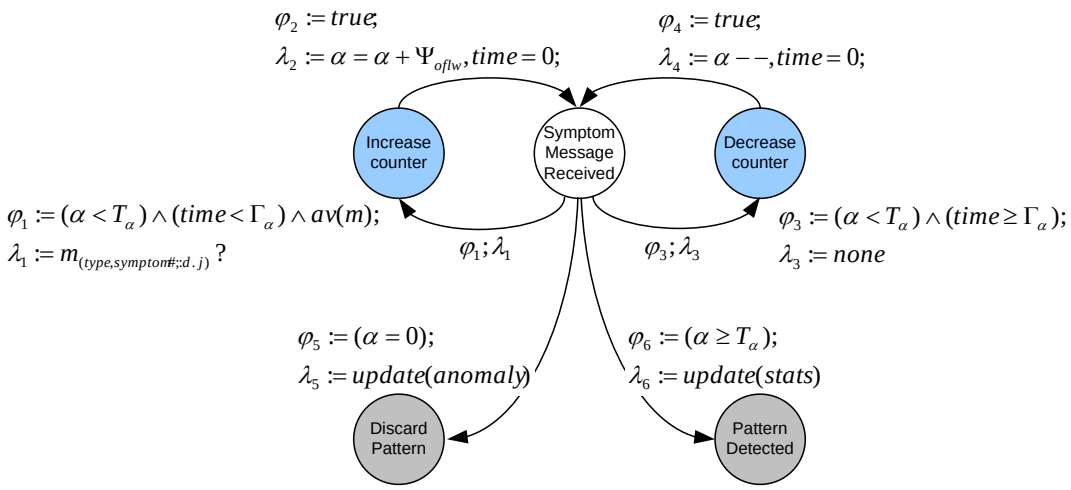

Figure 6.31: Out-of-Norm Analysis: Timed Automata for Job Borderline Fault Analysis

As depicted in Figure 6.29 the goal of the implemented analysis algorithm is the determination whether the sender or the receiver is faulty. This information is then correlated via the use of a hierarchical ONA to check for correlations in the space domain as previously discussed. This hierarchy of ONAs as shown in Figure 6.30 allows a determination whether a fault is delimited by physical or functional borders. The key advantage is that this way once defined ONAs can be used for multiple analysis processes and information from various sources can be correlated for operation on the distributed state of the system. As depicted in Figure 6.30 for each job an ONA with corresponding symptoms evaluates whether a job violates its (probabilistic) interface specification. Although we restrict the symptoms in the implementation to queue overflow detections and bursts, inclusion of interarrival and service time violations is straightforward. The result of this continuous evaluation process leads to a corresponding update of the cluster data structure `TD_decos_cluster`. Concurrently to this evaluation process that combines the diagnostic information for each job (i.e. job-local) a hierarchical ONA combines all results to check for correlations in the space domain (physical vs. functional) to improve the accuracy of the analysis.

The timed automata for the analysis of job borderline faults as depicted in Figure 6.31 implements a basic α-counter strategy. For each job with sending and receiving functionality two automata are needed. The reason for being able to keep each of the automata very simple is the possibility of deploying hierarchical ONAs. The Ψ_α value for increasing the α-counter depends on the type of symptom. While an overflow is a definite failure, a single burst is just an anomaly. Consequently, the α-counter value is increased with a higher value in case of overflows. Whenever the α-counter threshold value T_α is exceeded, the cluster data structure `TD_decos_cluster` is updated. For instance, if job 0 of the `CAN_COMFORT` DAS hosted on component 0 misses to consume the messages at the respective input ports of its link, the `receive_counter` and `borderline_counter` are updated:

```
cluster.component[0].DAS[CAN_COMFORT].job[0].receive_counter++;
cluster.component[0].DAS[CAN_COMFORT].job[0].borderline_counter++;
```

In analogy the `send_counter` is updated accordingly. Similar to previously described examples, the parameter Γ_α defines the length of the interval between decreasing the α-counter value in case no further symptom message is received. In case the α-counter value is decreased to 0, the fault pattern for the job is discarded. Nevertheless a corresponding update of the anomaly counter of the job in the cluster data structure `TD_decos_cluster` is made.

In case the analysis process has revealed a job software or borderline (i.e. configuration) fault, this information can then be correlated at the OEM with data gathered from a significant population of cars to identify design faults. This fleet analysis provides data for the improvement of the quality of the deployed hardware and software and also helps in minimizing warranty costs.

Parameter Settings

In the prototype implementation the following constants for the α-counter have been defined:

```
#define INIT_ALPHA_VALUE_SEND            20
#define INIT_ALPHA_VALUE_RECEIVE         20
#define T_ALPHA                         120
#define GAMMA_ALPHA                 0x5B8D80
#define PSI_ALPHA_SEND                   20
#define PSI_ALPHA_BURST                   3
#define PSI_ALPHA_RECEIVE                 5
```

Both, `INIT_ALPHA_VALUE_SEND` and `INIT_ALPHA_VALUE_RECEIVE` are used to increase the α-counter value after the first occurrence of a corresponding symptom (i.e. queue overflow at the input or output port). `T_ALPHA` defines the threshold and `GAMMA_ALPHA` the length of the time interval, after which the α-counter is decreased in case no further symptom is received at the input port of the analysis job. The penalty value `PSI_ALPHA_SEND` is added to the value of the α-counter of a queue overflow at a sender. In case a burst is detected a smaller value defined in `PSI_ALPHA_BURST` is added to the counter, since a burst is typically no definite violation of the temporal interface specification but rather an anomaly. In analogy, the penalty value `PSI_ALPHA_RECEIVE` is added to the value of the α-counter of a queue overflow at the receiver. This value is smaller compared to the value of `PSI_ALPHA_SEND`, since `PSI_ALPHA_RECEIVE` will be typically added multiple times (i.e. for every input port of the job).

Results

Figure 6.32 depicts an exemplary measurement curve for the α-counter value of a faulty sender. Each time the queue overflows, the penalty value is added to the

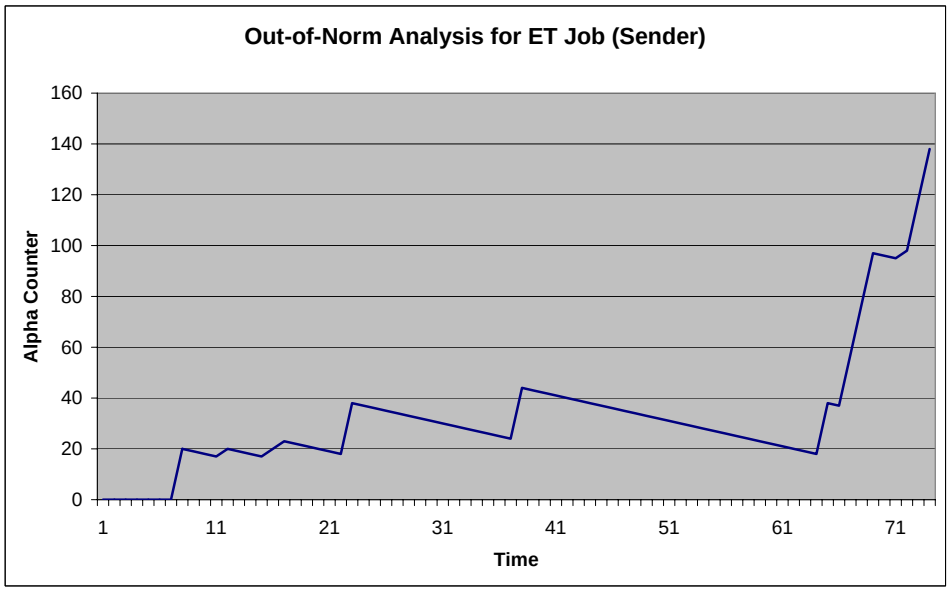

Figure 6.32: Out-of-Norm Analysis: Faulty Sender

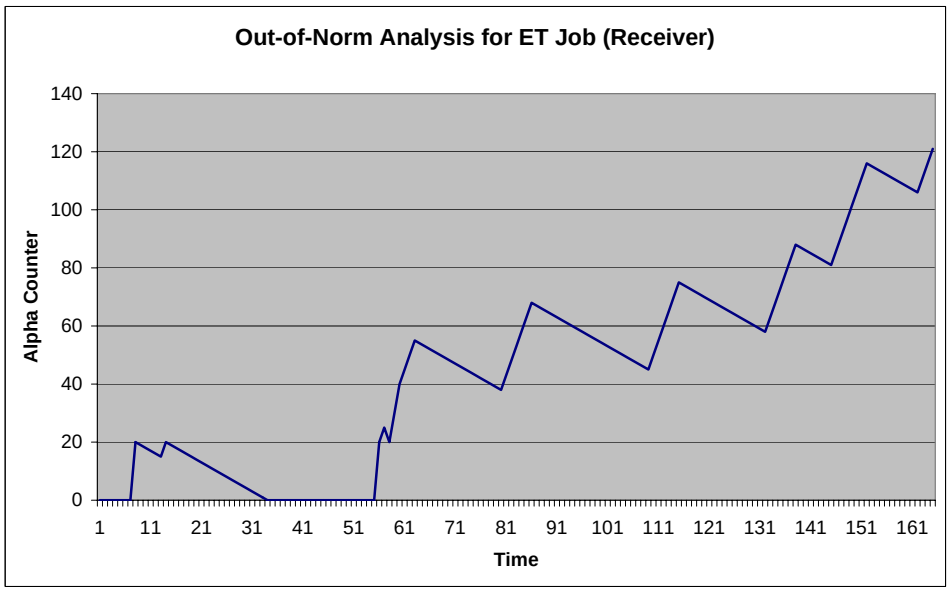

Figure 6.33: Out-of-Norm Analysis: Faulty Receiver

current α-counter value. In case a burst is detected, the α-counter value is increased by $\Psi_{\alpha-\text{burst}}$ (see for instance $t = 12$, $t = 15$ or $t = 72$).

On the other hand, Figure 6.33 shows an exemplary measurement curve for the out-of-norm analysis of a faulty receiver. The increase of the α-counter value depends on how many other event-triggered jobs send messages to the input port of the receiving job at the interval of job inactivity. The more queue overflows, the higher the increase of the α-counter value.

Interpretation of the Results

In comparison with diagnostic solutions for physical CAN systems, the diagnostic services operating on a virtual event-triggered network allow tracing of the job responsible for causing a violation of the interface specification. The lack of a global time base in physical CAN systems makes an analysis of correlated fault effects difficult. Furthermore, the sending of diagnostic messages with high priority can itself be the reason for delaying application messages resulting in violations of temporal interface specifications. In case of a low priority the sending of diagnostic messages without a globally synchronized timestamp can mislead the analysis process and subsequently lead to wrong maintenance actions. In addition, the limited error containment properties of CAN makes tracing the real fault source a difficult and error prone task.

Chapter 7

Selected Experiments and Results

In this chapter we present selected experiments to investigate the effects of fault-induced state changes on the distributed state of the core communication system in order to define fault patterns for diagnosis of systemic system failures. In particular, numerous fault injection experiments have been carried out to analyze the effects of external, internal, and borderline faults in order to identify systemic symptoms that allow for accurate diagnosis. Furthermore, the independence assumption for the implementation choices of the core time-triggered communication system of the DECOS architecture with respect to diagnosis is investigated. We thereby determine the type of information provided by the protocol mechanisms and communication controller that proves to be meaningful for maintenance purposes and then extrapolate this result to define generic systemic symptoms for time-triggered communication systems. In addition, the fault injection experiments demonstrate the benefits of deploying a dedicated virtual network not only for the transport of diagnostic information, but also as a key mechanism of a fault injection infrastructure whenever a system wide view on the effects of faults is required.

At first we describe the setup of the used fault injection framework. We have executed two fault injection campaigns according to the component fault classes of the maintenance-oriented fault model introduced in Section 5.3. One campaign uses an EMI disturbance generator for injecting internal/external component faults, the second campaign utilizes both, the EMI disturbance generator and TTTech disturbance node, for injecting fault representatives of the component borderline fault class.

7.1 Overview of the Fault Injection Framework

In the course of the fault injection campaigns we target only fault classes with respect to the component boundary, i.e. component internal, component external, and

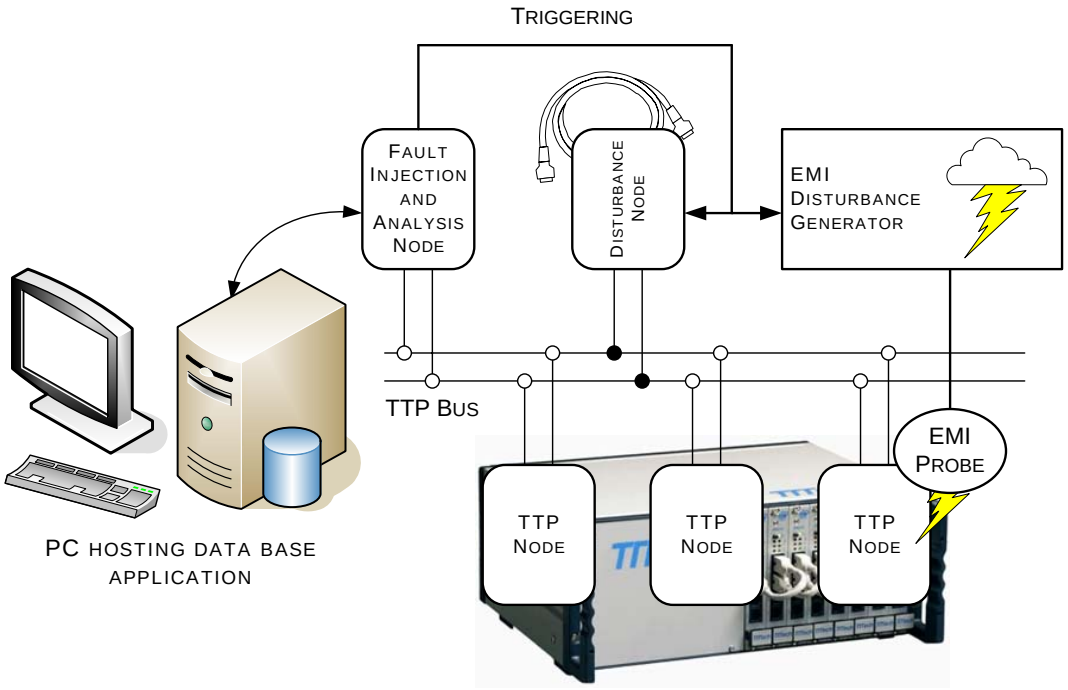

Figure 7.1: Setup of the Hardware Fault Injection Experiments

borderline faults. For this reason, a time-triggered cluster as the core communication system of the DECOS architecture is used. In the setup of the fault injection framework as depicted in Figure 7.1 we employ a TTP MPC555 cluster consisting of four nodes [TTT04a]. In the framework we utilize two hardware fault injection devices [HTI97], an EMI disturbance generator and a bus disturbance generator. While the latter is used to emulate borderline faults, the EMI generator is used to simulate both internal and external component faults. For a more detailed description of the fault injection setup refer to [Pau05]. During the fault injection campaigns the target system has only been exposed to radiated energy and not to power supply induced EMI.

One dedicated node in combination with a personal computer is responsible for triggering the experiments. Like in the DECOS architecture we employ a virtual diagnostic network, i.e. an overlay network on top of the core time-triggered network, for the dissemination of relevant information. This way, monitoring of the distributed state of the system is possible. Once the result of an experiment is transferred to the PC and stored in the database, the cluster is restarted for the following experiments to preclude any side effects of previous experiments.

Since we are not interested in testing the hardware for EMI susceptibility, but only interested in the effects EMI is causing on the distributed state of the system, it is intentional to make the device under investigation more susceptible to EMI. Therefore, a non automotive qualified version of the TTP nodes is used. By using the automotive qualified version of the hardware (by-wire-box), less effects are expected

Figure 7.2: The NSG 1025 Fast Transient/Burst Generator

in comparison with the version of the cluster for development.

7.1.1 Cluster Description

In the framework we use a TTP-Development Cluster based on four TTP-Powernodes mounted in a rack and with one TTP-Monitoring Node for downloading configuration data (i.e. the cluster schedule) and application code [TTT04a]. Each TTP-Powernode [TTT04b] is equipped with the TTP C2 controller AS8202NF (C2NF) and uses a Freescale MPC555 processor for the execution of application tasks. TTP-OS is used as the operating system [TTP05]. In addition to TTP, a broad variety of interfaces is supported: ISO 9141 (suitable for TTP/A, LIN, and ISO-K), CAN, digital I/O, and analog inputs. In the fault injection setup a bus topology is used.

7.1.2 The NSG 1025 Fast Transient/Burst Generator

The NSG 1025 fast transient/burst generator as depicted in Figure 7.2 is manufactured by Schaffner Instruments is used as an EMI-disturbance generator in our fault injection framework. The generator is designed for the use in lab and field-testing due to its capability of producing different types of disturbance signals. In particular, the ability to control the triggering of the EMI disturbances by software makes the system especially useful for lab testing, whenever a large number of experiments is required. Once the type of disturbances and parameters (i.e. the amplitude and frequency) has been selected, the fault injection campaign can be run automatically. All types of disturbances can be controlled using the trigger/gate port of the generator at TTL-voltage, thus the generator can be controlled using a standard output port of a node. Whenever no signal is provided, the generator transmits no disturbances.

A variety of probes of different shape and size can be used for the transmission of the disturbance signal. In addition, an oscilloscope can be connected to the

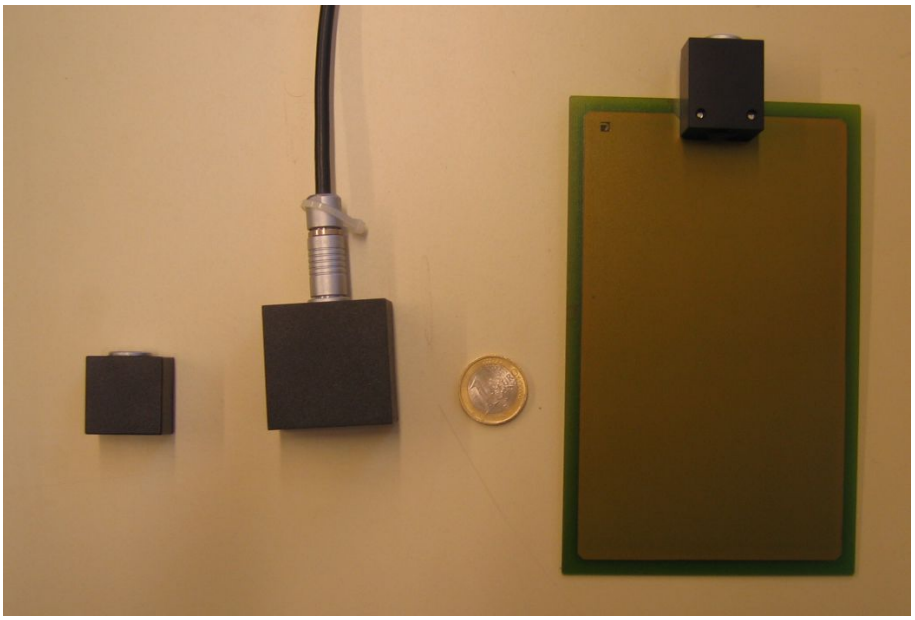

Figure 7.3: The Different Types of Probes of the EMI Testing Device (small (S), medium (M), and large (L))

EMI-device for monitoring purposes. This feature simplifies the development of the triggering software as it allows direct monitoring of the generated disturbance signals. The EMI-generator can generate the following type of disturbance signals:

- **Single pulses.** In single mode the EMI-disturbance generator produces fast transient pulses with a rise-time of 5ns, duration of 50ns and amplitudes from 225V to 4.4kV. Although the manual specifies 4.4kV as the maximum voltage, the generator can be adjusted to higher voltages up to 5.02kV.

- **Repeated single pulses (50Hz).** The form of the repeated pulses is the same as in single mode and the repetition-frequency of 50Hz is fixed. The amplitude can be selected either by choosing one of the pre-defined IEC-levels or continuously by using the potentiometer (e.g., for validation according to technical standards).

- **Burst disturbances.** By setting the EMI-generator to burst mode, burst disturbances of variable amplitude with a duration of 15ms that are repeated every 300ms are transmitted continuously.

  At burst mode the generation of disturbances using small values for period is permitted, but can bear problems as the generator pauses for 300ms before beginning the transmission of a new burst. If the value for period is smaller than this, the EMI-generator may not generate the full number of bursts requested.

Figure 7.4: The TTP-Disturbance Node

7.1.3 The TTTech TTP-Disturbances Node

The TTP-Disturbance Node [TTT05] (as illustrated in Figure 7.4) can be used to examine the system behavior of a TTP system that is subject to fault induced state changes. The node can be used in a running system (e.g., onboard a car) or with a test setup in the laboratory. This way the laboratory experiments can be replicated onboard the target system. In the test setup the disturbance node is used to study the effects of faults on system behavior, in particular to produce borderline faults. A key advantage of the disturbance node is the reproducibility of the test cases, since the triggering of the experiments can be synchronized with the TDMA cluster scheme. The disturbance node is easily configured using XML description files and can be triggered directly by the analysis node via the output port pins. The disturbance node can be used to inject a variety of classes of faults, thus allowing analyzing the behavior at the physical, logical, and application layer. For instance the following faults can be injected:

- Loss of transmissions
- Short-circuits to VCC and ground
- Mismatched termination of bus wires
- Noise burst
- Loss of specific data (reproducible experiments possible)
- Wrong Cyclic Redundancy Code (CRC)

7.2 Analysis of Component External Failures using Electromagnetic Interference (EMI)

The use of EMI is widely accepted as a solid fault injection technique for analyzing the effects of non destructive external faults on electronic devices. The following campaign investigates the fault induced state changes of the TTP cluster under EMI in order to define diagnostic mechanisms.

7.2.1 Hypotheses

The first hypothesis is a statement about the restart rate (also component membership) for diagnosis. As elaborated on in Section 3.5 the restart rate of a component is frequently considered as a suitable indication whether a transient internal or external fault is affecting a component. The experiments will determine the restart rates of the nodes of the cluster when exposed to EMI, as a non destructive external fault injection technique, and give valuable information on how to use the restart rate for analysis. Furthermore, on the basis of the restart rate of the nodes, a fault pattern can be defined, which is a prerequisite for the definition of the symptoms and the parametrization of the subsequent analysis algorithms.

Hypothesis 1 *A single restart of a TTP MPC555 node (PN212) is justifiably considered as an effective symptom for the identification of external component faults.*

Subclaim 1 *When a node is exposed to EMI, Invalid Frames are detected on the network for the duration of the fault injection.*

Subclaim 2 *When a node is exposed to EMI, Incorrect Frames are detected on the network for the duration of the fault injection.*

Subclaim 3 *When a node is exposed to EMI, Null Frames are detected on the network for the duration of the fault injection.*

Subclaim 4 *When a node exhibits a crash failure due to EMI, it reintegrates into the cluster within a bounded interval of time.*

The detection of correlated node failures is crucial in reducing the fault-not-found ratio in today's embedded systems. The second hypothesis states that in the DECOS architecture such correlated failures due to EMI can be identified and forwarded for subsequent analysis. Note, that correlated node failures are not within the single fault hypothesis [Kop04]. It is expected that by using a bus topology with local guardians, the effects of EMI induced state changes in a component may physically propagate on the bus and cause adjacent nodes to fail. This hypothesis is thus a statement of investigating physical effects on the bus, not error propagation via insidious messages.

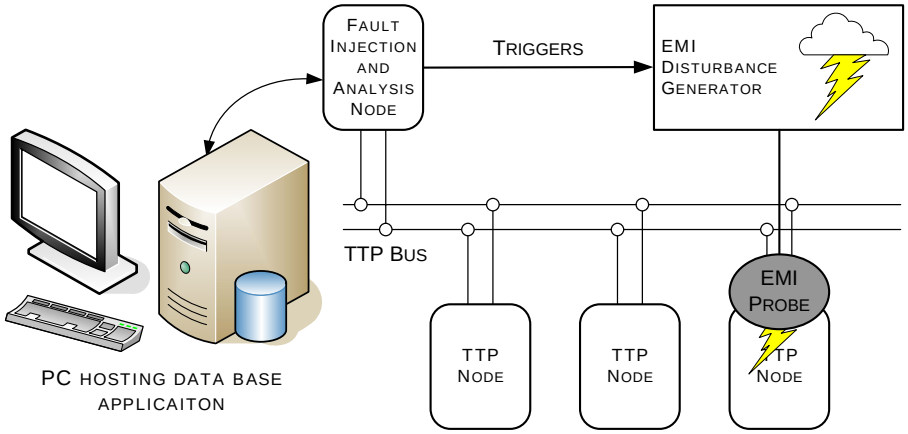

Figure 7.5: Setup for the EMI Fault Injection Experiments for Analyzing Component Failures

Hypothesis 2 *In case of fail-silent node failures of two nodes, a correlation of these multiple node failures due to the effects of EMI can be identified by the diagnostic subsystem on the basis of diagnostic messages sent on the virtual diagnostic network indicating the frame status of the nodes.*

The DECOS architecture is designed to be independent from the core communication network, as long as the core services are provided. The third hypothesis evaluates whether this assumption holds in case of EMI. If this is not the case, then for other choices for the underlying time-triggered core network, such as FlexRay or Time-Triggered Ethernet, adapted detection strategies need to be devised and implemented.

Hypothesis 3 *The DECOS diagnostic services are independent from the implementation technology of the underlying time-triggered core communication system.*

7.2.2 Experimental Setup

As illustrated in Figure 7.5 the EMI testing device is used for evaluating the effects of component external faults on the state of the distributed system. For this reason the probe is directly placed on one node of the cluster. The experiments are executed using different probe types, EMI disturbance modes, and voltage levels.

7.2.3 Results

Table 7.1 shows the results of a total of 80,000 EMI fault injection experiments targeting the C2 controller. The type field denotes whether a random single pulse (SPRand) or if a single pulse at the sending slot (SPSlot) of the node under investigation has been injected. For these experiments the small probe has been used (see Figure 7.3). For both types, SPRand and SPSlot, voltage levels of 2000 and

Type	Probe	Position	Voltage	# of Experiments
SPRand2S	S	C2	2000	10000 / 10000
SPRand4S	S	C2	4000	10000 / 10000
SPSlot2S	S	C2	2000	10000 / 10000
SPSlot4S	S	C2	4000	10000 / 10000
Type	**Membership Loss**	**Restart Rate**		**Null Frames**
SPRand2S	486 / 959	1,0 / 0,40		486 / 959
SPRand4S	752 / 6274	1,0 / 0,84		752 / 6274
SPSlot2S	5672 / 6206	1,0 / 0,89		5672 / 6206
SPSlot4S	1062 / 9988	1,0 / 0,74		1062 / 9988

Table 7.1: Results of 80,000 EMI Experiments (Target: C2 controller, Mode: Single)

Type	Probe	Position	Voltage	# of Experiments
SPSlot4L	L	Node	4000	5000
SPRand4L	L	Node	4000	5000
Type	**Membership Loss**	**Restart Rate**		**Null Frames**
SPSlot4L	502	0,51		502
SPRand4L	298	0,21		298

Table 7.2: Results of 10,000 EMI Experiments (Target: Node, Mode: Single)

4000 Volts have been investigated. As expected, the higher the voltage used in the EMI experiments, the more likely the node failed. It is worth recognizing, that only Correct or Null Frames have been received by all other nodes in the cluster. Furthermore, the restart rate (i.e. the rate the node restarted itself after a failure) has always been 100% when using the hardware watchdog of the MPC555. Without the explicit use of the hardware watchdog a restart rate significantly below 100% has been recorded (numbers listed in table: "MPC555 watchdog/controller only").

Table 7.2 presents the results of a total of 10,000 experiments using a probe targeting the complete TTP node (with disabled MPC555 watchdog). Both, random single pulses and single pulses at the sending slot of the node under investigation have been injected with a voltage level of 4000 Volts. When executing the SPSlot4L using the large probe, 101 failures (i.e. 10 simultaneous and 91 single failures) of the node next to the node under analysis have been detected due to propagation of the EMI fault effects on the bus system. In case of the SPRand4L campaign 327 failures (i.e. 28 simultaneous and 299 single failures) have been detected. Similar to the results of Table 7.1, the node always failed in a fail-silence way (i.e. sending out either Correct Frames or Null Frames).

Table 7.3 lists the results of 10,000 fault injection experiments (Burst4L) targeting the complete node using the large (L) probe for injecting 2 ms EMI bursts in the slot of the node under investigation (i.e. node 3). In contrast to previous campaigns, the burst mode causes not only failures of the node (and restarts after a lost membership), but also channel failures resulting in Invalid and Incorrect Frames. As shown, node 0 monitored 6444 Null Frames on both channels, 9 Invalid frames (wrong encoding) on

Type	Probe	Position	Voltage
Burst4L	L	Node	4000
Node #	**Null Frames**	**Invalid Frames**	**Incor. Frames**
0 / Channel 0	6444	9	12
0 / Channel 1	6444	29	12
1 / Channel 0	6442	12	12
1 / Channel 1	6442	9	12
2 / Channel 0	6357	19	13
2 / Channel 1	6382	26	13

Table 7.3: Results of 10,000 EMI Experiments (Target: Node, Mode: Burst)

channel 0 and 29 Invalid Frames on channel 1, and 12 Incorrect Frames (i.e. incorrect CRC) on both channels. Node 1 monitored 6442 Null Frames on both channels, 12 Invalid Frames on channel 0 and 9 Invalid Frames on channel 1, and 12 Incorrect Frames on both channels. Finally, node 2 monitored 6357 Null Frames on channel 0 and 6238 Null Frames on channel 1, 19 Invalid Frames on channel 0 and 26 Invalid Frames on channel 1, and 13 Incorrect Frames on both channels. Since node 2 is next to node 3 in the bus configuration of the fault injection experiments, it can be concluded that the effects of the EMI disturbances have propagated.

7.2.4 Interpretation and Discussion

In the following we discuss the results derived from the above presented fault injection campaign:

- **Hypothesis 1 has not been falsified.**

  Subclaim 1 has not been falsified. A total of 37 Incorrect Frames (i.e. wrong CRC) has been consistently detected on both channels of the TTP cluster when bursts are randomly injected.

  Subclaim 2 has not been falsified. A total of 104 Invalid Frames (i.e. wrong encoding) has been detected when bursts are randomly injected. In contrast to the Incorrect Frames, the number of detected Invalid Frames per channel was different.

  Subclaim 3 has not been falsified. A total of 100,000 EMI experiments has revealed that in approximately 40% of all runs, the node under investigation crashed in a fail silent way, i.e. Null Frames have been consistently detected on the bus by all other nodes in the cluster.

  Subclaim 4 has not been falsified. The restart rate presented in Table 7.1 and Table 7.2 shows that whenever a node has exhibited a crash failure, the node restarted and reintegrated into the cluster (including the virtual diagnostic network) within 130 ms to 3490 ms. This large variation in time can be explained due to the mechanisms triggering the restart of node. When built-in mechanisms of the TTP protocol trigger the restart, the blackout time of a node is

relatively small. However, in case of a host failure, then the blackout time of the node is significantly longer. This is due to the fact, that the automatically generated code of the used TTP tools prohibits a user defined setting of the MPC555 watchdog configuration registers. Since the watchdog can only be configured at startup (and this is performed by non user modifiable tool generated code), the watchdog is automatically set to the maximum delay value. The experiments have shown that there are likely scenarios in which the node does not restart after an EMI fault when the hardware watchdog provided by the MPC555 is disabled. However, when in addition to the built-in mechanisms the hardware watchdog of the MPC555 is enabled, a restart rate of 100% has been investigated.

Whenever a node fails to restart after a non-destructive transient external fault, the node will be classified as being permanent faulty by the analysis process, although a restart of the affected node during the next cycle of operation (e.g., shutting the engine off and on again in a car), such a node will be operating as specified again. Consequently, for maximum availability of nodes and to minimize the possibility of declaring functional nodes as being permanently faulty, a strategy should be implemented of restarting nodes whenever, either the host or the communication controller of the node fails. Note that in the used TTP cluster, only the host can reset the communication controller, but the communication controller cannot reset the host, whenever a lost membership is detected. Although this way the application may enter a safe state, we believe that a never-give-up strategy of immediately restarting the node would be a better choice. Otherwise, not only the system is operating in a degraded service mode (e.g., only two nodes of a TMR system are operating), but also prone to misjudgement by the diagnostic services.

It is obvious that such a strategy requires a new system design strategy, since the interface state needs to be periodically transmitted on the network to allow fast state re-synchronization. This strategy of state-aware system design, has the drawback of requiring additional bandwidth for the dissemination of interface state information. However, with emerging technologies such as Time-Triggered Ethernet [Kop05] today's bandwidth problems are significantly extenuated.

As a result of the experiments it can be concluded that in the majority of the experiments a node perturbed by EMI fails in a fail-silent way, i.e. whenever the node under investigation was disturbed by the injected fault, the node failed to send a frame in its assigned slot, i.e. a Null Frame was detected due to a transient hardware failure. Subsequently, the node is classified as non operational since the node loses its membership. This has always been consistently detected by all nodes in the cluster. Thus, the experiments have shown, that the inclusion of the restart rate is justifiably considered in literature (e.g., [BCGG97, BCGG00]) as an effective symptom that should be used as input for analysis algorithms.

- **Hypothesis 2 has not been falsified.**

  Results of the fault injection experiments described in Table 7.2 indicate that by using a bus topology with local guardians, the effects of EMI induced state changes in a component may physically propagate on the bus and cause adjacent nodes to fail. A total of 38 correlated node failures has been detected out of a total of 10,000 experiments.

  In these cases the correlated loss of the membership of two nodes within the cluster has been consistently detected by the remaining operational nodes. Via timestamping of the diagnostic messages encoding the membership loss (i.e. als recognized via Null Frames as the frame status on both physical channels), the analysis subsystem can easily deduce such a correlation of a fault causing dual node failures. In the setup a 16 bit timestamp provided by the TTP/C global time service is used. Therefore, the virtual diagnostic network is required to transport this information to the analysis node before the occurrence of an overflow to allow for a meaningful detection of correlated fault effects. No violation of this timing assumption has been detected during the experiments.

  As a result, in case of a bus topology spatial proximity information between nodes should be encoded into the analysis algorithms to exploit this knowledge for improving the accuracy of the result in case of correlated node failures. When using a star topology, this problem should be solved, since a star configuration is superior in handling EMI faults. However, additional fault injection campaigns are necessary to validate this claim.

- **Hypothesis 3 has not been falsified.**

  Since the vast majority of the injected faults have resulted in crash failures, it can be concluded that a binary information (i.e. the node is operational or not) about the functionality of a node is sufficient as input in order to classify the experienced failures according to the maintenance-oriented fault model. In TTP, such a binary classification is provided by the membership service (i.e. providing a consistent cluster wide view on the operational status of each node in the cluster). In case no membership service is provided by the core technology, the cross-checked frame status indicating missing sending activity (i.e. a Null Frame in TTP) is an alternative choice. This independence of the core communication controller is important, since the DECOS architecture is designed to be suitable for a variety of underlying time-triggered core networks. This way the same or very similar symptoms can be used also in the case of using other time-triggered core communication system such as Time-Triggered Ethernet [Kop05] or FlexRay [Fle04]. As derived from the experiments the membership information or the restart of a node (especially when applying an according never-give-up strategy of continuously restarting non operational nodes) is a reproducible symptom valid for any time-triggered core architecture.

7.3 Analysis of Component Borderline Failures using Electromagnetic Interference (EMI) and the Disturbance Node

This campaign aims at investigating the behavior of the TTP cluster under borderline faults generated by the disturbance node and by the EMI disturbance generator. The experiments are used to inject borderline faults according to the maintenance-oriented fault model and monitor the fault induced interface state changes of the TTP cluster. Depending on the results of this fault injection campaign, systemic symptoms for the detection and analysis of component borderline faults can be devised.

7.3.1 Hypotheses

The possibility to identify symptoms that allow discrimination between component and bus/connector failures is a prerequisite for applying the maintenance-oriented fault model. Therefore, different fault patterns need to be identified that allow definition of symptoms for the different fault classes. The first hypothesis states that using the frame status information provided by TTP C2 controller as part of the PN212 node allows for discrimination between component and bus/connector faults. Furthermore, the result will determine whether a binary classification of the frame status is sufficient for the analysis of borderline faults or whether a finer granularity is required.

Hypothesis 1 *The value of the frame status field provided by the TTP/C controller of the deployed TTP PN212 cluster allows distinguishing between component internal/external faults and borderline faults.*

Identification of the exact location of a failure is an important step in keeping associated maintenance/warranty costs low. Whenever, the location cannot be determined, there is always the possibility of changing fully operational parts of the system. The following hypothesis states that in our setup employing a bus topology the location (affected node) of the failure can be identified.

Hypothesis 2 *When using the TTP PN212 cluster with a bus topology, the exact location of a borderline (bus) failure can be determined on the basis of the frame status information distributed on the virtual diagnostic network.*

Since bandwidth is typically considered in industry as a scarce resource, the transport of diagnostic information on only one channel is an alternative option in order to save bandwidth.

Hypothesis 3 *The transport of diagnostic information via a virtual diagnostic network on top of only one physical channel of the TTP cluster allows detection of all injected borderline faults.*

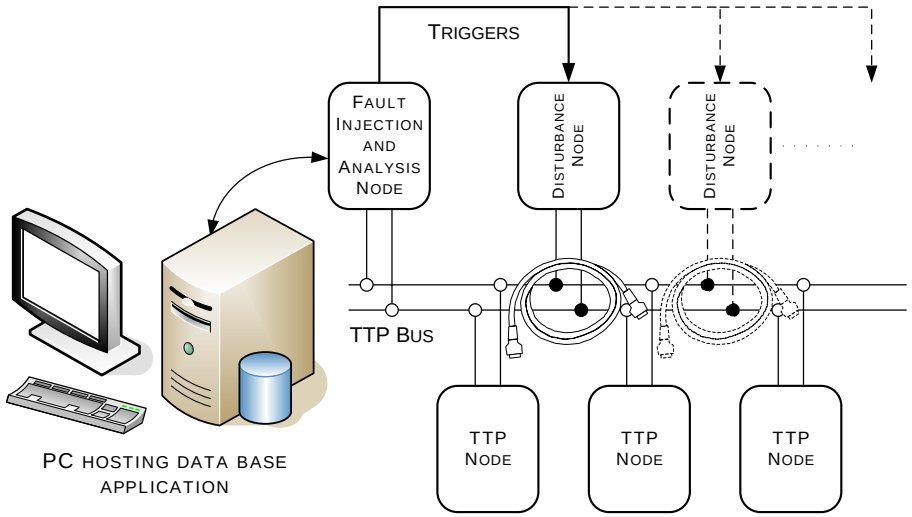

Figure 7.6: Setup for the Borderline Fault Injection Experiments

7.3.2 Experimental Setup

As illustrated in Figure 7.6 the disturbance node is used to simulate component borderline faults. The disturbance node can be inserted into the bus between any two TTP nodes, since for fault injection no modifications of the cluster schedule are required (i.e. in the used configuration, the disturbance node does not actively generate valid TTP frames).

In addition, for analyzing the effects of heavy EMI disturbances on the bus a similar configuration like in the previously described EMI injection campaigns has been used. In this configuration the probe of the EMI testing device is placed on the bus to affect more than one node (depending on the time interval specified for the injection campaign).

7.3.3 Results

Table 7.4 shows the results of the EMI experiments targeting the bus of the TTP cluster. In order to strengthen the effects of the EMI on the bus, the twisted pair wire has been split up to allow a higher failure rate. For these 13000 experiments the voltage has been set to 500 Volts and the L probe has been used to disturb channel 1 (bus line TTH) with 15 ms bursts. In the first column the node of the cluster that is observing either an Invalid Frame or Null Frame on the respective channel 0 or 1 is listed. For each row the “accused node” entry contains the node of the cluster that is suspected to be faulty. The data indicates that perturbing a channel of the bus using the EMI probe causes a significant number of Invalid Frames on channel 1. Null Frames are only received when the node fails to operate and restarts (i.e. is losing its membership).

During the fault injection campaign using the disturbance node as shown in

Type	Probe	Position	Voltage	# of Experiments
Burst0.5L	L (15 ms)	Channel 1	500	13000
Node #	**Channel**	**Accused Node**	**Invalid Frames**	**Null Frames/ML**
0	0	1	0	5
0	1	1	93	5
0	0	2	0	6
0	1	2	48	6
0	0	3	0	24
0	1	3	1908	24
1	0	0	0	0
1	1	0	213	0
1	0	2	0	2
1	1	2	58	2
1	0	3	0	24
1	1	3	1936	24
2	0	0	0	0
2	1	0	689	0
2	0	1	0	1
2	1	1	417	1
2	0	3	0	24
2	1	3	1997	24
3	0	0	0	0
3	1	0	1975	0
3	0	1	0	5
3	1	1	1961	5
3	0	2	0	6
3	1	2	1889	6

Table 7.4: Results of EMI Experiments (Target: Bus, Mode: Burst (15 ms))

Type	Busline TTL	Busline TTH	# of Exp.	Time	Detected Frames
DN1	GND	GND	1000	40 ms	Invalid (Correct)
DN2	VCC	VCC	1000	40 ms	Invalid
DN3	GND	VCC	1000	40 ms	Null (Invalid)
DN4	VCC	GND	1000	40 ms	Invalid
DN5	GND	-	1000	40 ms	Invalid
DN6	VCC	-	1000	40 ms	Invalid
DN7	-	GND	1000	40 ms	Invalid
DN8	-	VCC	1000	40 ms	Null (Invalid)

Table 7.5: The Fault Injection Experiments using the Disturbance Node

Figure 7.7 a total of 8000 experiments has been investigated to analyze the effects of short circuits on the bus of the cluster. Table 7.5 lists the eight different types of experiments that have been executed. Each of the eight configurations of short

Figure 7.7: Fault Injection using the Disturbance Node

Node #	Accused Node	Invalid Frames	Null Frames	Correct Frames
0	1	0	1000	1000
0	2	0	1000	1000
0	3	0	1000	1000
1	0	2000	1000	1000
1	2	0	1000	1000
1	3	0	1000	1000
2	0	2000	1000	1000
2	1	0	1000	1000
2	3	0	1000	1000
3	0	2000	1000	1000
3	1	0	1000	1000
3	2	0	1000	1000

Table 7.6: Results for Campaign DN3 and DN8

circuits to GND and VCC lasted for 40 ms starting at the beginning of the slot of node 0 on channel 0. For each type a total of 1000 experiments has been injected and analyzed.

During the runs DN2, DN4, DN5, DN6, and DN7 the nodes monitor Invalid Frames for the duration of 4 TDMA rounds (a TDMA round has been configured to 10 ms).

In Table 7.6 the results for the fault injection campaign DN3 are presented (the

Node #	Accused Node	Invalid Frames	Null Frames	Correct Frames
0	1	1000	0	1000
0	2	1000	0	1000
0	3	1000	0	1000
1	0	1000	0	1000
1	2	1000	0	1000
1	3	1000	0	1000
2	0	1000	0	1000
2	1	1315	0	1315
2	3	1000	0	1000
3	0	1000	0	1000
3	1	1000	0	1000
3	2	1000	0	1000

Table 7.7: Results for Campaign DN1

experiments for campaign DN8 showed the same results). In all 1000 experiments all nodes of the cluster detect in slot of node 0 (in which the fault injection starts) an Invalid Frame. Then, for 3 TDMA rounds Null Frames are consistently monitored and again an Invalid Frame is sent at the end of the experiment. The reason for the two Invalid Frames at the beginning and end of the experiment is due to the truncation of the frame disseminated by node 0 by the disturbance node.

Table 7.7 presents the results of campaign DN1. The results are similar compared to the results of the campaigns DN2, DN4, DN5, DN6, and DN7. However, besides monitoring Invalid Frames, node 2 detects also Correct Frames during fault injection due to characteristics of the used physical layer. For a more detailed discussion refer to [Pau05].

7.3.4 Interpretation and Discussion

By interpreting the previously described results of the fault injection campaign the following results have been determined:

- **Hypothesis 1 has not been falsified.**

  As the results of the fault injection experiments presented in Table 7.4 and 7.5 have revealed, by monitoring the frame status field of both physical channels, a discrimination between component and bus/borderline failures is possible.

  While EMI faults affecting the node typically result in the failure of the node and thus to the detection of Null Frames on the bus with subsequent membership loss, when disturbing the bus using the disturbance node, typically Invalid Frames are detected. A similar fault pattern can be investigated when using the EMI device to perturb the bus. In most cases the EMI causes the detection of Invalid Frame types. Furthermore, borderline faults affect only one channel

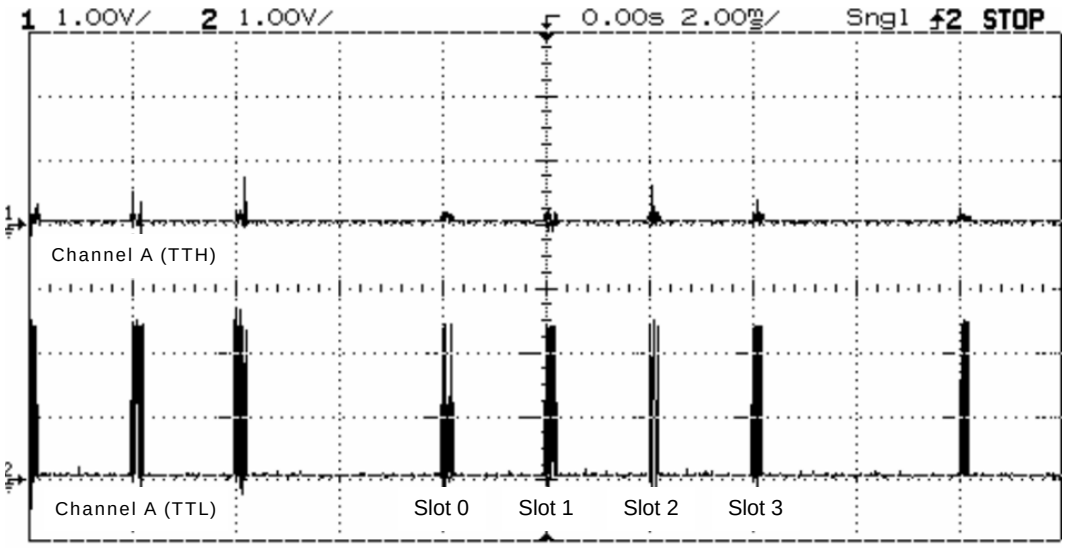

Figure 7.8: A Short Circuit injected on Channel A of the TTP Bus.

of the core communication system, while node failures manifest themselves via Null Frames on both channels.

This fact, that in case of hardware problems using a bus topology, Invalid Frames are significantly more likely detected compared with the results of the EMI campaign targeting the node allows defining the occurrence of Invalid Frames as a good choice for symptoms of borderline/bus faults.

Consequently, the experiments have revealed that a binary classification scheme is not sufficient for the discrimination between fault classes according to the maintenance-oriented fault model. TTP provides a classification scheme for frames that discriminates between Null, Incorrect, Invalid, Tentative and Correct frames (see also Section 6.3). For the DECOS independence assumption with respect to the underlying time-triggered core architecture, this results implicates that an abstract interface, onto the hardware-specific controller registers of TTP, Time-Triggered Ethernet or FlexRay can be mapped, needs to be defined. The results of the experiments suggest to provide at least a message classification that allows to discriminate between correct frames, frames with wrong CRC, frames with wrong encoding, and null frames.

- **Hypothesis 2 has been falsified.**

  In our fault injection setup using a bus topology the exact location of a short circuit cannot be traced, since the bus is disturbed for the duration of the fault injection. Figure 7.8 shows an oscilloscope screenshot illustrating an exemplary scenario, where only Invalid Frames are received due to a short injected via the disturbance node. During the fault injection time interval, the frames sent by all nodes on channel A are affected. Pinpointing the node responsible for this failure is impossible.

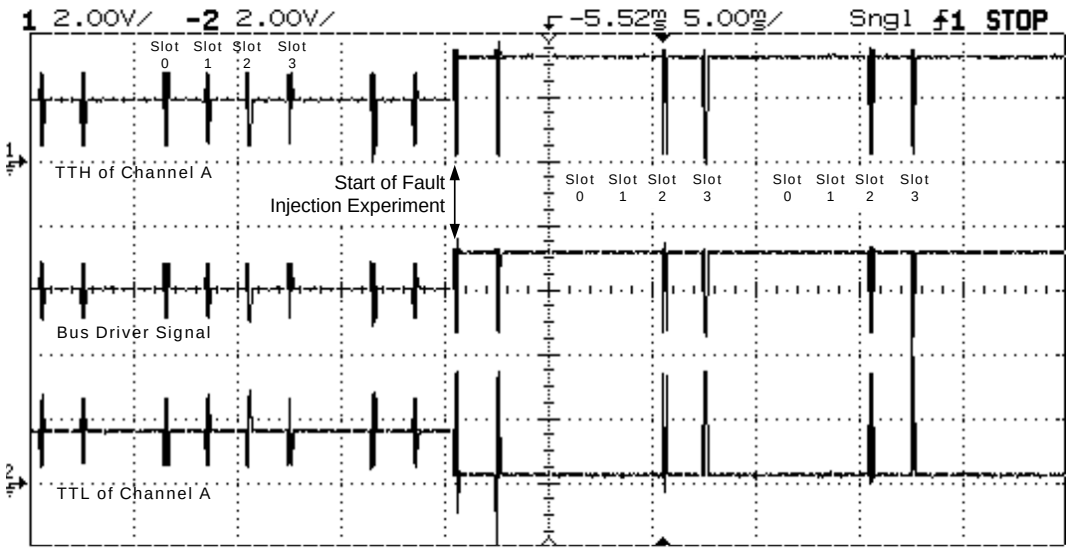

Figure 7.9: A Bus Failure on Channel A dividing the Cluster.

Note, that the results are only valid for the employed physical layer of the TTP cluster used in the fault injection framework. In case of point-to-point connections, as used in a star topology with a central bus guardian, a significantly better detection capability is expected. However, additional fault injection campaigns are necessary to validate this claim.

- **Hypothesis 3 has been falsified.**

  Although sending different non safety-critical messages in a node's slot per channel allows for doubling the available bandwidth, in case of diagnosis this strategy is problematic. Once the channel fails, no more diagnostic information can be distributed and processed. The results of the experiments listed in Table 7.4, and Table 7.5 indicate that this situation can occur.

  Figure 7.9 shows such a scenario where channel A of the bus is divided into two parts by a fault induced by the disturbance node. Here, node 0 and 1 would classify node 2 and 3 as faulty and vice versa. In case only one channel is used, insufficient information would be available at the analysis subsystem. In case of a central diagnostic solution, one part of the system would therefore be classified as being permanently faulty. For this reason both channels should be used for the dissemination of diagnostic information. In order to save bandwidth alternative strategies, such as forwarding of symptoms regarding channel A on channel B and vice versa are possible. In addition, by splitting the remaining messages between the outgoing queues of the event-triggered overlay network and thus decrease the average load per channel, the latency can also be reduced. However, such a strategy has the penalty of increased complexity and requires additional processor time.

Chapter 8

Conclusion

The main contribution of this thesis is the development of diagnostic architectural services as part of an integrated time-triggered architecture in order to tackle prevalent maintenance problems industry is currently facing. The DECOS architecture, as developed in a research project within the Sixth European Framework Programme, aims at unifying the respective benefits of federated and integrated architectures. On top of stable and validated core services that include a time-triggered transport service, clock synchronization, fault isolation, and consistent component diagnosis, a set of high-level services such as the virtual networks service or the gateway service facilitates application development. As part of these high-level services, the diagnostic and maintenance service aims at reducing service times and associated warranty costs.

In accordance with hardware trends, emphasis has been set on the detection and analysis of transients. Since transients last, by definition, only for a short interval of time, typical BISTs that are scheduled periodically are not sufficient. Consequently, only by continually evaluating the interface state of the constituting physical and functional elements of the integrated system, all failures and anomalies can be detected and analyzed. This is a prerequisite to enable the diagnostic services to discriminate between transients resulting from internal and external fault sources. While the latter require no maintenance actions, system internal faults can only be corrected by replacement. Such a discrimination is possible by including the frequency and correlated effects of experienced failures into the analysis process. In combination with the error containment properties of the DECOS architecture and the inclusion of knowledge about the system structure, accurate diagnostic services can be realized.

The Maintenance-Oriented Fault Model

As part of the conceptual design we have introduced a maintenance-oriented fault model that establishes the conceptual foundation of the diagnostic services of the

DECOS integrated architecture. The fault model takes the component-based nature of today's distributed embedded systems into account stops the recursion of the fault-error-failure chain at a level suitable for maintenance. According to this model each experienced failure is classified with respect to the field replaceable units of the system. The fault model also reflects the physical and functional entities of the system to allow classification between hardware and software faults. For components we discriminate between external, borderline, and internal faults, while for jobs (i.e. software modules) a classification of faults into external, borderline, and inherent (software vs. attached I/O) is realized. By applying the fault model during the design of the deployed diagnostic mechanisms, it is ensured that only those checks are realized that help in answering the question whether a particular FRU needs to be replaced.

The presented fault model is also very useful for defining the test cases for validation purposes using fault injection techniques. Thereby, each test case relates to one of the specified fault classes. The fault injection campaigns as part of this thesis were designed after these considerations.

Operation on the Distributed State

In the context of diagnosis, integrated architectures exceed comparable federated systems by the possibility to operate on the distributed state to reveal correlated failures. By taking the physical and functional structure of an integrated system into account a more accurate reasoning about the nature of a fault affecting the system in case of failure is possible. Since integrated architectures, such as the DECOS architecture, overcome the prevalent "1 Function - 1 ECU" design philosophy, a discrimination between software and hardware faults is feasible. In combination with the inter-component and inner-component error containment mechanisms provided by the basic and high-level services, this strategy allows to trace correlated system anomalies back to the FCR responsible for the experienced system behavior. In contrast to the internal component states, the distributed state can be independently checked. Such a detection is thus much more trustworthy than any internal check that cannot be verified. Consequently, this strategy provides the foundation for solving prevalent diagnostic problems by taking a holistic view, in contrast to diagnostic systems operating only on the local internal state (e.g., like most OBD system currently deployed in the automotive industry).

We introduced Out-of-Norm Assertions (ONAs) as the primary diagnostic mechanism following these principles that are operating on the distributed state to detect correlated malfunctions. ONAs take the characteristics of faults in the time, value and space domain into account in order to discriminate between different types of faults according to the maintenance-oriented fault model that are affecting the operation of the distributed system. Since ONAs are specified on the interface state, mutual error detection of interface state variables is performed. The key concept behind the cross-checking principle is that a failure of the sender manifests itself as an

error in the interface state of the receiver (with respect to the interface specification of the sender).

When shifting to time-triggered architectures, existing event-triggered legacy application will not be replaced instantly. ONAs can also be employed, if the specification includes imprecise temporal specifications such as probabilistic timing assumptions used for event-triggered communication. Consider for instance the specification of inter-arrival times of messages in event-triggered communication systems. Normally, this information is expressed via probability distributions, thus a sharp line that allows a classification into correct and incorrect cannot be simply drawn. By introducing the concept of ONAs we provide a mechanism that allows to classify such system anomalies in case of insufficient interface specifications over time.

The implementation of the ONA concept in the prototype DECOS cluster has shown promising results. In particular, the inclusion of the time and space domain besides value information has proved to be important for accurate diagnosis. As supported by fault injection data, the use of timing information allows for detection of correlated failures. In addition, when taking taking space information, i.e. knowledge about the physical and functional structure of the system, into account, the detection of software faults as required in integrated architectures is possible.

Diagnostic and Maintenance Services for an Integrated Time-Triggered Architecture

In order to integrate diagnosis into the development process, a framework supporting the definition of the diagnostic services has been introduced. Such a framework also encourages the developers to precisely specify the diagnostic checks early in the design and to treat diagnosis not as an addendum but as an integral part of all development phases. Due to the framework, design faults can be avoided by automatically transforming the ONA specifications into executable code that can be executed as part of the high-level services.

To support separation of concerns, systemic diagnosis is decoupled from application level diagnosis. This way the efforts of both, the system and application designers, can be reduced. The application-specific diagnostic services can be parameterized according to the knowledge of the application developers. By contrast, the systematic diagnostic techniques (e.g., identification of component hardware failures) are independent of a particular application context and need not be covered by the respective application-specific diagnosis strategy (e.g., plausibility checks). In addition, a revalidation of the systemic diagnosis mechanisms by the OEM is not necessary if the coverage of the deployed mechanisms has been validated and can be reproduced deterministically.

In the framework timed automata are used for the specification of ONAs, which proved to be a solid and powerful mechanism. Furthermore, the transformation of

timed automata into executable code is straightforward. In the prototype implementation for each of the fault class of the maintenance-oriented fault model we have provided an example showing the feasibility of using the proposed specification method. In particular, we specified and implemented application specific symptoms for time-triggered and event-triggered jobs and systemic symptoms for component external, internal and borderline faults.

Extensive fault-injection experiments have been carried out to investigate to which extent the DECOS diagnostic services allow abstraction from the target platform (e.g., TTP, FlexRay, Time-Triggered Ethernet). The campaigns have revealed which of the provided TTP-specific controller information can be used for the specification of systemic symptoms. As a result it can be summarized that a restart of a node is a suitable indicator for the detection of internal/external component failures. In combination with algorithms that process the frequency of this failure mode as their input, a judgment on the type of fault affecting the node is possible. In combination with frame status information provided by all nodes of the cluster, a classification according to the maintenance-oriented fault model is realistic.

Preclusion of Probe Effects via Encapsulation of the Diagnostic Services

Integrated architectures promise substantial technical and economic benefits in the development of distributed embedded real-time systems. However, the benefits of integrated architectures can easily be extenuated by increasing the complexity of the system. In particular, with respect to diagnosis and maintenance some challenges emerge. The higher the integration, the more easily services will interfere without an error containment strategy. In the context of the diagnosis it is therefore important that the diagnostic services are not the reason for any failure. In particular, architectural means must be realized to ensure that no probe effect will be introduced given the economic constraints imposed by industry.

A necessary part of an integrated diagnostic infrastructure is the continuous monitoring and subsequent dissemination of diagnostic information to allow tracing of experienced failures back to the origin. Such a dissemination must not introduce any additional temporal uncertainty in the communication system that negatively affects the real-time traffic. In this thesis we propose a virtual diagnostic network on top of the core time-triggered physical network. The a priori assigned bandwidth guarantees independence of temporal behavior from the communication activities of other virtual networks. By exploiting the virtual network high-level service of the DECOS architecture no additional wiring and contact points are introduced. Thus, not only a cost effective solution from an economic point of view is realized, but also no additional faults can be introduced by the diagnostic services itself. This way the introduction of a probe effect is precluded by design. Extensive fault injection campaigns targeting the DECOS core communication network have shown that transport of diagnostic information via a dedicated event-triggered virtual diagnostic

network is robust and reliable. Furthermore, the reserved bandwidth for diagnosis can be parameterized according to the system constraints.

In order to facilitate certification of the DECOS architecture, the diagnostic services are not part of the safety-critical subsystem and are only deployed in the non safety-critical part of the system. Since only elementary interfaces are used for the sending of diagnostic messages on the virtual diagnostic network, no back-pressure flow control can negatively affect any application. Furthermore, by realizing the analysis jobs in an encapsulated diagnostic DAS, the temporal and spatial partitioning services, i.e. inner-component error containment services, provided by the DECOS architecture ensure that the effects a software design fault of an analysis job cannot propagate at the partition boundary and negatively affect jobs of other DASs.

Bibliography

[AAS79] J.M. Adams, J. Armstrong, and M. Smartt. Assertional checking and symbolic execution: An effective combination for debugging. In *Proceedings of the 1979 annual conference*, pages 152–156. ACM Press, 1979.

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

[Ada84] E.N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.

[Aer91] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Specification 651: Design Guide for Integrated Modular Avionics*, November 1991.

[Aer93] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Specification 624: Design Guidance for Onboard Maintenance System*, August 1993.

[Aer96] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Report 624: No Fault Found - A case study*, April 1996.

[Aer03] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Specification 653-1 (Draft 3): Avionics Application Software Standard Interface*, July 2003.

[AKC90] M Ahuja, A. Kshemkalyani, and T. Carlson. A basic unit of computation in distributed systems. In *Proc. 10th Intl. Conf. Distributed Computing Systems (ICDCS-10)*, Paris, France, May 28-June 1 1990. IEEE.

[ALR01] A. Avizienis, J.C. Laprie, and B. Randell. Fundamental concepts of dependability. Research Report 01-145, LAAS-CNRS, Toulouse, France, April 2001.

[ALS88] A. Avizienis, M. Lyu, and W. Schutz. In search of effective diversity: A six-language study of fault-tolerant flight control software. In *The Eighteenth International Symposiun on Fault Tolerant Computing*, pages 15–22, 1988.

[AM02] S. Amberkar and B. Murray. Diagnostic strategies for advanced automotive systems. In *Convergence International Congress & Exposition On Transportation Electronics*, Detroit, MI, USA, October 2002. SAE.

[Apt81] K.R. Apt. Ten years of Hoare's logic: A survey part I. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):431–483, 1981.

[AriCE] Aristotle. *Physics*. 350 B.C.E. Translated by R.P. Hardie and R.K. Gaye.

[ASBT03] A. Ademaj, H. Sivencrona, G. Bauer, and J. Torin. Evaluation of fault handling of the time-triggered architecture with bus and star topology. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, pages 123–132, June 2003.

[Avi75] A. Avizienis. Fault-tolerance and fault-intolerance: Complementary approaches to reliable computing. In *Proceedings of the international conference on Reliable software*, pages 458–464, 1975.

[Bar01] J. Barkai. Vehicle diagnostics–are you ready for the challenge? In *Proceedings of Automotive & Transportation Technology (ATT) Congress & Exhibition*, volume 5. SAE, October 2001.

[Bau01] G. Bauer. *Transparent Fault Tolerance in a Time-Triggered Architecture*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2001.

[Bax97] C. Baxter. Fly as you view [in-flight entertainment]. *IEE Review*, 43(3):118–121, May 1997.

[BB98] P.G. Bishop and R.E. Bloomfield. A methodology for safety case development. In *Proceeedings of the Safety-critical Systems Symposium*, Birmingham, UK, February 1998.

[BBBCD00] F. Bachmann, L. Bass, C. Buhman, and S. Comella-Dorda. Technical concepts of component-based software engineering. Technical Report 008, CMU/SEI, Pittsburgh, May 2000.

[BBD+00] D. Beal, E. Bianchi, L. Dozio, S. Hughes, P. Mantegazza, and S. Papacharalambous. RTAI: Real-Time Application Interface. *Linux Journal*, April 2000.

[BCGG97] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, and F. Grandoni. Discriminating fault rate and persistency to improve fault treatment. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 354–362, Washington - Brussels - Tokyo, June 1997. IEEE.

[BCGG00] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, and F. Grandoni. Threshold-based mechanisms to discriminate transient from intermittent faults. *IEEE Transactions on Computers*, 49(3):230–245, March 2000.

[BCV91] R.W. Butler, J.L. Caldwell, and B.L. Di Vito. Design strategy for a formally verified reliable computing platform. In *Proceedings of the 6th Annual Conference on Systems Integrity, Software Safety and Process Security*, pages 125–133, June 1991.

[Ber02] I. Berger. Can you trust your car? *IEEE Spectrum*, 39(4):40–45, April 2002.

[BG00] J. Berwanger and M. Peller R. Griessbach. Byteflight a new protocol for safety critical applications. In *Proceedings of the FISITA World Automotive Congress*, Seoul, 2000.

[Bir03] S. Birch. Pre-safe headlines S-Class revisions. *Automotive Engineering International*, pages 15–18, January 2003.

[BJ01] T. Brotherton and T. Johnson. Anomaly detection for advanced military aircraft using neural networks. In *Proceedings of the IEEE Aerospace Conference*, volume 6, pages 3113–3123. IEEE, March 2001.

[BJPW99] A. Beugnard, J. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, July 1999.

[BKS03] G. Bauer, H. Kopetz, and W. Steiner. The central guardian approach to enforce fault isolation in a time-triggered system. In *Proceedings of the 6th International Symposium on Autonomous Decentralized Systems (ISADS 2003)*, pages 37–44, Pisa, Italy, April 2003.

[BOK+99] A. Brown, D. Oppenheimer, K. Keeton, R. Thomas, J. Kubiatowicz, and D.A. Patterson. Istore: introspective storage for data-intensive network services. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 32–37, March 1999.

[Bos91] Robert Bosch Gmbh, Stuttgart, Germany. *CAN Specification, Version 2.0*, 1991.

[Bos02] Robert Bosch GmbH, editor. Vieweg Verlag, Braunschweig/Wiesbaden. *Autoelektrik Autoelektronik*, 4th edition, 2002.

[BP00] G. Bauer and M. Paulitsch. An investigation of membership and clique avoidance in TTP/C. In *Proceedings of 19th IEEE Symposium on Reliable Distributed Systems*, pages 118–124, Nürnberg, Germany, October 2000.

[Bre01] E. Bretz. By-wire cars turn the corner. *IEEE Spectrum*, 38(4):68–73, April 2001.

[BRO+02] G. Brahnmath, R. Raje, A. Olson, B. Bryant, M. Auguston, and C. Burt. A quality of service catalog for software components. In *Proceedings of the Southeastern Software Engineering Conference*, Huntsville, Alabama, 2002.

[BT93] D. Briere and P. Traverse. Airbus A320/A330/A340 electrical flight controls – a family of fault-tolerant systems. In *The Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 616–623, Aerospatiale, Toulouse, June 1993.

[BW98] A. W. Brown and K. C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, September/October 1998.

[CDK94] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. International Computer Science Series. Addison-Wesley, Reading, MA, USA, second edition, 1994.

[CDLS99] R.G. Cowell, A.P. David, S.L. Lauritzen, and D.J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Springer-Verlag New York, Secaucus, NJ, USA, 1999.

[CG02] C.R. Carlson and J.C. Gerdes. Practical position and yaw rate estimation with GPS and differential wheelspeeds. In *Proceedings of 6th International Symposium on Advanced Vehicle Control*, pages 481–486, Hiroshima Japan, September 2002.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[Cha91] E. Charniak. Bayesian networks without tears. *AI Magazine*, 12(4):50–63, 1991.

[Col99] R. Collinson. Fly-by-wire flight control. *Computing & Control Engineering Journal*, 10:141–152, August 1999.

[Con02] C. Constantinescu. Impact of deep submicron technology on dependability of VLSI circuits. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 205–209. IEEE, 2002.

[Cri91] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[CS96] O. Ciupke and R. Schmidt. Components as context-independent units of software. In *Proc. of ECOOP*, 1996.

[CW84] E. Chow and A. Willsky. Analytical redundancy and the design of robust failure detection systems. *IEEE Transactions on Automatic Control*, 29(7):603–614, July 1984.

[CWGW04] R. Conatser, J. Wagner, S. Ganta, and I. Walker. Diagnosis of automotive electronic throttle control systems. *Control Engineering Practice*, 12(1):23–30, January 2004.

[CWH00] M. Campione, K. Walrath, and A. Huml. *The Java Tutorial: A Short Course on the Basics*. Addison-Wesley, third edition edition, 2000.

[Dei02] A. Deicke. The electrical/electronic diagnostic concept of the new 7 series. In *Convergence International Congress & Exposition On Transportation Electronics*, Detroit, MI, USA, October 2002. SAE.

[DoD98] DoD. *Military Handbook, Electronic Reliability Design Handbook (MIL-HDBK-338)*. US Department of Defense, 1998.

[Don00] B. Donham. Electromagnetic interference from passenger-carried portable electronic devices. *Aero*, 10:13–19, March 2000.

[DP02] D.D. Dylis and M.G. Priore. A new reliability assessment technique for aging electronic systems. In *Proceedings of the 6th Joint FAA/DoD/NASA Aging Aircraft Conference*, September 2002.

[DvdG00] M.J. Druzdzel and L.C. van der Gaag. Building probabilistic networks: Where do the numbers come from? *IEEE Transactions on Knowledge and Data Engineering*, 12(4):481–486, 2000.

[DW99] D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, Mass., first edition, 1999.

[Eas93] S. Easterbrook. Negotiation and the role of the requirements specification. In P. Quintas, editor, *Social Dimensions of Systems Engineering: People, processes, policies and software development*, pages 144–164. Ellis Horwood, 1993.

[Elm02] W. Elmenreich. *Sensor Fusion in Time-Triggered Systems*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2002.

[EW03] R. Etzold and C. Wüst. Fahrzeug steht, Kunde läuft. *Spiegel*, 40:170–172, 2003.

[Fel94] T. Felke. Application of model-based diagnostic technology on the Boeing 777 airplane. In *Proceedings of the 13th AIAA/IEEE Digital Avionics Systems Conference (DASC)*, pages 1–5, October 1994.

[FH01] C. Furse and R. Haupt. Down to the wire [aircraft wiring]. *IEEE Spectrum*, 38(2):34–39, February 2001.

[Fle01] W.J. Fleming. Overview of automotive sensors. *IEEE Sensors Journal*, 1(4):296–308, December 2001.

[Fle03] LIN Consortium. *LIN Specification Package Revision 2.0*, September 2003.

[Fle04] FlexRay Consortium. BMW AG, DaimlerChrysler AG, General Motors Corporation, Freescale GmbH, Philips GmbH, Robert Bosch GmbH, and Volkswagen AG. *FlexRay Communications System Protocol Specification Version 2.0*, July 2004.

[Flo67] R.W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Proceeding of the Symposium in Applied Mathematics*, volume 19, pages 19–32, 1967.

[FLP85] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[FO00] N.E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, August 2000.

[For95] US Air Force. Aircraft mishap investigation handbook for electronic hardware. Technical Report WL-TR-95-4004, Materials Directorate, Wright Laboratory, Air Force Material Command, January 1995.

[Gai86] J. Gait. A probe effect in concurrent programs. *Software Practice and Experience*, 16(3):225–233, March 1986.

[GAM+02] P. Gil, J. Arlat, H. Madeira, Y. Crouzet, T. Jarboui, K. Kanoun, T. Marteau, J. Duraes, M. Vieira, D. Gil, J.C. Baraza, and J. Gracia. *Fault Representativeness*. LAAS-CNRS, Toulouse, France, 2002. Deliverable (ETIE2) of the European Project Dependability Benchmarking Dbench (IST-2000-25425).

[GCB98] F. Grandoni, S. Chiaradonna, and A. Bondavalli. A new heuristic to discriminate between transient and intermittent faults. In *Proceedings of the Third IEEE International Symposium on High-Assurance Systems Engineering*, pages 224–231, 1998.

[GCF+95] J. Gertler, M. Costin, X. Fang, Z. Kowalczuk, M. Kunwer, and R. Monajemy. Model based diagnosis for automotive engines – algorithm development and testing on a production vehicle. *IEEE Transactions on Control Systems Technology*, 3(1):61–69, March 1995.

[Ger88] J.J. Gertler. Survey of model-based failure detection and isolation in complex plants. *IEEE Control Systems Magazine*, 8(6):3–11, December 1988.

[GF03] A.A. Ghobbar and C.H. Friend. Evaluation of forecasting methods for intermittent parts demand in the field of aviation: a predictive model. *Computers & Operations Research*, 30(14):2097–2114, December 2003.

[GFL+02] P. Giusto, A. Ferrari, L. Lavagno, J.-Y. Brunel, E. Fourgeau, and A. Sangiovanni-Vincentelli. Automotive virtual integration platforms: why's, what's, and how's. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 370–378, September 2002.

[GH02] H. Guo and W. Hsu. A survey on algorithms for real-time bayesian network inference. In *Proceedings of the joint AAAI-02/KDD-02/UAI-02 workshop on Real-Time Decision Support and Diagnosis Systems*, Edmonton, Alberta, Canada, 2002.

[GIJ+02] M.-C. Gaudel, V. Issarny, C. Jones, H. Kopetz, E. Marsden, N. Moffat, M. Paulitsch, D. Powell, B. Randell, A. Romanovsky, R. Stroud, and F. Taiani. Final version of the DSoS conceptual model. *DSoS Project (IST-1999-11585) Deliverable CSDA1*, December 2002. Available as Research Report 54/2002 at http://www.vmars.tuwien.ac.at.

[Goo00] D. L. Goodman. Prognostic techniques for semiconductor failure modes. Technical report, Ridgetop Group, Inc., Tucson, Arizona, USA, 2000.

[Gra86] J. Gray. Why do computers stop and what can be done about it? In *Proceedings of the 5th Symposium on Reliablity in Distributed Software and Database Systems*, January 1986.

[Gru93] T.R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[GS91] D. Galler and G. Slenski. Causes of aircraft electrical failures. *Aerospace and Electronic Systems Magazine*, 6(8):3–8, August 1991.

[GS02] R. Gao and A. Suryavanshi. Diagnosis from within the system [built-in test]. *IEEE Instrumentation & Measurement Magazine*, 5(3):43–47, September 2002.

[GSOS01] B.S.M.C. Galvao, G. Santos, H. Onusic, and L.F.P. Sant'Anna. Electromagnetic environmental measurements in specific populated areas of Brazil. In *Proceedings of the IEEE International Symposium on Electromagnetic Compatibility*, volume 1, pages 106–110, August 2001.

[GW02] R.A. George and C.J. Wang. Vehicle E/E system integrity from concept to customer. In *Convergence International Congress & Exposition On Transportation Electronics*, Detroit, MI, USA, October 2002. SAE.

[Ham03] R. Hammett. Flight-critical distributed systems: design considerations [avionics]. *IEEE Aerospace and Electronic Systems Magazine*, 18(6):30–36, June 2003.

[Hec95] D. Heckerman. A tutorial on learning with bayesian networks. Technical Report MSR-TR-95-06, Microsoft Research, Redmond, Washington, USA, 1995.

[Hei50] H.W. Heinrich. *Industrial Accident Prevention - A Scientific Approach.* McGraw-Hill Book Company Inc, New York, 1950.

[Hei03] H.D. Heitzer. Development of a fault-tolerant steer-by-wire steering system. *Auto Technology*, 4:56–60, April 2003.

[HHG90] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, pages 169–180, October 1990. Published as Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices, volume 25, number 10.

[Hil99] M. Hiller. Error recovery using forced validity assisted by executable assertions for error detection: an experimental evaluation. In *Proceedings of the 25th EUROMICRO Conference*, volume 2, pages 105–112. IEEE, September 1999.

[Hil00] M. Hiller. Executable assertions for detecting data errors in embedded control systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*, pages 24–33. IEEE, June 2000.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa00] T. Hoare. Assertions. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Proceeding of the Second International Conference on Integrated Formal Methods*, LNCS 1945, pages 1–2, Dagstuhl Castle, Germany, January 2000. Springer-Verlag Heidelberg.

[Hol92] I. M. Holland. Specifying reusable components using contracts. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 287–308, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[HPOS05] B. Huber, P. Peti, R. Obermaisser, and C. El Salloum. Using RTAI/LXRT for partitioning in a prototype implementation of the DECOS architecture. In *Proceedings of the Third International Workshop on Intelligent Solutions in Embedded Systems (WISES)*, May 2005.

[HR02] The Hansen Report on Automotive Electronics, November 2002. Portsmouth NH USA, `www.hansenreport.com`.

[HS00] P. Herzum and O. Sims. *Business Component Factory.* OMG Press, 2000.

[HT98] G. Heiner and T. Thurner. Time-triggered architecture for safety-related distributed real-time systems in transportation systems. In *Proceedings of the Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, pages 402–407, June 1998.

[HTI97] M. Hsueh, T.K. Tsai, and R.K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, April 1997.

[IEE99] *IEEE standard methodology for reliability prediction and assessment for electronic systems and equipment*, January 1999. IEEE Std 1413-1998.

[ISO95] International Standardization Organisation, ISO 7637. *Road vehicles – Electrical disturbances from conduction and coupling*, 1995.

[JM02] E. Juliussen and P. Magney. Telematics technology trends. *Automotive Engineering International*, pages 54–61, September 2002.

[Jon01] W.D. Jones. Keeping cars from crashing. *IEEE Spectrum*, 38(9):40 – 45, September 2001.

[JW02] L.C. Jaw and D.N. Wu. Anomaly detection and reasoning with embedded physical model. In *Proceedings of the IEEE Aerospace Conference*, volume 6, pages 6–3073–6–3081. IEEE, March 2002.

[KB03] H. Kopetz and G. Bauer. The time-triggered architecture. *IEEE Special Issue on Modeling and Design of Embedded Software*, January 2003.

[KBJ00] L. Kaufman, S. Bhide, and B. Johnson. Modeling of common-mode failures in digital embedded systems. In *Proceedings of the Reliability and Maintainability Symposium*, pages 350–357, Los Angeles, CA, USA, 2000. IEEE Press.

[KBS05] H. Kopetz, G. Bauer, and W. Steiner. *The Industrial Communication Technology Handbook*, chapter Dependable Time-Triggered Communication, pages 12–1 – 12–16. CRC Press, February 2005.

[KGR91] H. Kopetz, G. Grünsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. *Dependable Computing for Critical Applications (A. Avizienis and J. C. Laprie, Eds.), pp.411-29, Springer-Verlag, Wien, Austria*, 1991.

[KHTP99] K. Kimseng, M. Hoit, N. Tiwari, and M. Pecht. Physics-of-failure assessment of a cruise control module. *Microelectronics Reliability*, 39:1423–1444, 1999.

[KJ00] L.F. Kaufmann and B.W. Johnson. Modeling of common-mode failures in digital embedded systems. In *Proc. of the Reliability and Maintainability Symposium*, Los Angeles, Cal., 2000.

[KJ03] I. Kim and Y. Jung. Using bayesian networks to analyze medical data. In P. Perner and A. Rosenfeld, editors, *Lecture Notes in AI*, volume 2734, pages 317–327. Spinger-Verlag, 2003.

[KJH00] Z. Kiziltan, T. Jonsson, and B. Hnich. On the definition of concepts in component based software development. Technical report, Mälardalen University, Västras, Sweden, 2000.

[KK90] H. Kopetz and K. H. Kim. Temporal uncertainties in interactions among real-time objects. In *Proceedings of Ninth Symposium on Reliable Distributed Systems*, pages 165–174, October 1990.

[Kle75] L. Kleinrock. *Queuing Systems Volume I: Theory.* John Wiley and Sons, New York, 1975.

[KN97] H. Kopetz and R. Nossal. Temporal firewalls in large distributed real-time systems. In *Proceedings of IEEE Workshop on Future Trends in Distributed Computing*, Tunis, Tunesia, 1997. IEEE Press.

[KO87] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real time systems. *IEEE Transactions on Computers*, August 1987.

[KO02] H. Kopetz and R. Obermaisser. Temporal composability. *Computing & Control Engineering Journal*, 13:156–162, August 2002.

[Kop92] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proceedings of 12th International Conference on Distributed Computing Systems*, Japan, June 1992.

[Kop95] H. Kopetz. Why time-triggered architectures will succeed in large hard real-time systems. In *Proceedings of the 5th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, Cheju Island, Korea, August 1995.

[Kop97] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.

[Kop98] H. Kopetz. Component-based design of large distributed real-time systems. *Control Engineering Practice - A Journal of IFAC*, 6:53–60, 1998.

[Kop99a] H. Kopetz. Elementary versus composite interfaces in distributed real-time systems. In *Proceedings of ISADS'99*, Tokyo, Japan, March 1999.

[Kop99b] H. Kopetz. *Specification of the TTP/C Protocol*. TTTech, Schönbrunner Straße 7, A-1040 Vienna, Austria, July 1999. Available at `http://www.ttpforum.org`.

[Kop00] H. Kopetz. Software engineering for real-time: A roadmap. *Proceedings of the 22nd International Conference on Future of Software Engineering (FoSE) at ICSE 2000, 4th - 11th June 2000, Limerick, Ireland*, Jun. 2000.

[Kop01] H. Kopetz. The temporal specification of interfaces in distributed real-time systems. In *Proceedings of the EMSOFT 2001, Tahoe City California, USA*, pages 223–236, October 2001.

[Kop03] H. Kopetz. Fault containment and error detection in the time-triggered architecture. In *Proceedings of the Sixth International Symposium on Autonomous Decentralized Systems*, April 2003.

[Kop04] H. Kopetz. The fault hypothesis for the time-triggered architecture. In *Proceedings of the IFIP World Computer Congress*, 2004.

[Kop05] H. Kopetz. The design of time-triggered Ethernet. In *Proceedings of the 8th IEEE International Symposium on Object-oriented Real-time distributed Computing*, Seattle, Washington, May 2005.

[KOPS04] H. Kopetz, R. Obermaisser, P. Peti, and N. Suri. From a federated to an integrated architecture for dependable embedded real-time systems. Technical Report 22, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.

[Kru98] P. Kruchten. Modeling component systems with the unified modeling language. In *Proc. of International Workshop on Component Based Engineering*, 1998.

[KS03a] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 51–60, May 2003.

[KS03b] H. Kopetz and N. Suri. On the limits of the precise specification of component interfaces. In *Proceedings of the Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*, pages 26–27, 2003.

[KWFT88] R.M. Kieckhafer, C.J. Walter, A.M. Finn, and P.M. Thambidurai. The maft architecture for distributed fault tolerance. *IEEE Transactions on Computers*, 37:398–404, April 1988. Allied Signal Corp., Columbia, MD.

[KWS00] H. Kim, A.L. White, and K.G. Shin. Effects of electromagnetic interference on controller-computer upsets and system stability. *IEEE Transactions on Control Systems Technology*, 8(2):351–357, March 2000.

[LA90] P.A. Lee and T. Anderson. *Fault Tolerance Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer Verlag, 1990.

[Lad97] P. Ladkin. Electromagnetic interference with aircraft systems: why worry? Report rvs-j-97-03, Bielefeld University, Faculty of Technology, 1997.

[Lan02] E. Lansinger. Software support for telematics. *Automotive Engineering International*, pages 32–34, October 2002.

[Lap92] J.C. Laprie. *Dependability: Basic Concepts and Terminology*. Springer Verlag, Vienna, Austria, 1992.

[LC02] L. Lev and P. Chao. Down to the wire. Technical report, Cadence Design Systems, Inc., San Jose, CA, USA, 2002.

[Lee98] D.B. Lee. In-flight entertainment-getting from wishlist to reality. In *Proceedings of the 17th Digital Avionics Systems Conference*, volume 2, pages G16/1 –G16/8, October 1998.

[Lee01] E. A. Lee. Embedded software from concurrent component models. *ACM SIGPLAN Notices*, 36(8), 2001.

[LH94] J.H. Lala and R.E. Harper. Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE*, 82:25–40, January 1994.

[LH02] G. Leen and D. Heffernan. Expanding automotive electronic systems. *Computer*, 35(1):88–93, January 2002.

[LHD99] G. Leen, D. Heffernan, and A. Dunne. Digital networks in the automotive vehicle. *Computing & Control Engineering Journal*, 10(6):257–266, December 1999.

[LK00] G. Lui-Kwan. In-flight entertainment: the sky's the limit. *IEEE Computer*, 33(10):98–101, October 2000.

[LKYZ00] Y.-H. Lee, D. Kim, M. Younis, and J. Zhou. Scheduling tool and algorithm for integrated modular avionics systems. In *Proceedings of the 19th Digital Avionics Systems Conference*, volume 1, pages 1C2/1–1C2/8, Philadelphia, PA, USA, October 2000.

[LM01] R.R. Lutz and I.C. Mikulski. Evolution of safety-critical requirements post-launch. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, pages 222–227. IEEE, August 2001.

[LPS01] B. Littlewood, P. Popov, and L. Strigini. Modeling software design diversity: a review. *ACM Comput. Surv.*, 33(2):177–208, 2001.

[LS04] J. Leohold and C. Schmidt. Communication requirements of future driver assistance systems in automobiles. In *Proceedings of the IEEE International Workshop on Factory Communication Systems*, pages 167–174, September 2004.

[Lut93] R.R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 126–133. IEEE, January 1993.

[LXR+02] L. Li, J. Xie, O.M. Ramahi, M. Pecht, and B. Donham. Airborne operation of portable electronic devices. *IEEE Antenna's and Propagation Magazine*, 44(4):30–39, August 2002.

[Mah00] E. Maher. No-fault finder. *Aviation Maintenance MAgazine*, May 2000.

[Mat88] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, France, October 1988. Elsevier.

[McB93] J.W. McBride. Electrical contact and connectors in automotive systems. In *Proceedings of the IEE Colloquium on Connectors on Vehicles*, pages 3/1–3/7, November 1993.

[Mey92] B. Meyer. Applying design by contract. *Computer*, 25(10):40–51, October 1992.

[MIS98] MISRA. *Guidelines for the use of the C language in vehicle based software.* The Motor Industry Software Reliability Association (MISRA), April 1998.

[MN88] B.M. McMillin and L.M. Ni. Executable assertion development for the distributed parallel environment. In *Proceedings of the Twelfth International Computer Software and Applications Conference (COMPSAC 88)*, pages 284–291. IEEE, October 1988.

[MNW98] G. McCall, D. Newman, and D. Ward. Guidance on automotive software development in relationship to emc. In *Proceedings of the IEE Colloquium on Electomagnetic Compatibility of Software*, pages 8/1–8/5, November 1998.

[MOS02] MOST Cooperation, Karlsruhe, Germany. *MOST Specification Version 2.2*, November 2002.

[MPG02] S. Mishra, M. Pecht, and D.L. Goodman. In-situ sensors for product reliability monitoring. In *Proceedings of SPIE*, volume 4755, pages 10–19, 2002.

[MRS+02] M. Mateos, P. Robin, S. Sauvage, V. Joloboff, G. Madhusudan, and Y. Bennani. Environment for evolutionary automotive diagnosis. In *Convergence International Congress & Exposition On Transportation Electronics*, Detroit, MI, USA, October 2002. SAE.

[MS02] I. Moir and A. Seabridge. *Aircraft systems: mechanical, electrical, and avionics subsystems integration.* Professional Engineering Publishing, London, UK, 2nd edition, 2002.

[MS03] I. Moir and A. Seabridge. *Civil Avionics Systems.* Professional Engineering Publishing, London, UK, 2003.

[MSZ01] P. Müller, C. Stich, and C. Zeidler. Components @ Work: Component Technology for Embedded Systems. In *27th Euromicro Workshop on Component-Based Software Engineering Workshop*, Warsaw, Poland, September 2001. IEEE Computer Society.

[MT89] M. D. Mesarovic and Y. Takahara. *Abstract Systems Theory*, chapter 3. Springer-Verlag, 1989.

[MT02] R.A. Maxion and K.M.C. Tan. Anomaly detection in embedded systems. *IEEE Transactions on Computers*, 51(2):108–120, February 2002.

[Mur01] K.P. Murphy. An introduction to graphical models. Technical report, University of British Columbia, Vancouver, USA, 2001.

[Nob92] I.E. Noble. EMC and the automotive industry. *Electronics & Communication Engineering Journal*, 4(5):263–271, October 1992.

[Nob94] I.E. Noble. Electromagnetic compatibility in the automotive environment. *Science, Measurement and Technology*, 141(4):252–258, July 1994.

[Nor96a] E. Normand. Single-event effects in avionics. *IEEE Transactions on Nuclear Science*, 43(2):461–474, April 1996.

[Nor96b] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43(6):2742–2750, December 1996.

[Nyb02] M. Nyberg. Model-based diagnosis of an automotive engine using several types of fault models. *IEEE Transactions on Control Systems Technology*, 10(5):679–689, September 2002.

[Obe04] R. Obermaisser. *Event-Triggered and Time-Triggered Control Paradigms – An Integrated Architecture.* Real-Time Systems Series. Kluwer Academic Publishers, November 2004.

[OM02] T. Ogawa and H. Morozumi. Diagnostic trends for automotive electronic systems. In *Convergence International Congress & Exposition On Transportation Electronics*, Detroit, MI, USA, October 2002. SAE.

[OMG02] Object Management Group, Needham, MA 02494, U.S.A. *The Common Object Request Broker: Architecture and Specification*, July 2002.

[OMG03] Object Management Group (OMG). *Smart Transducers Interface, Version 1.0*, January 2003.

[OP05] R. Obermaisser and P. Peti. Specification and execution of gateways in integrated architectures. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Catania, Italy, September 2005. IEEE.

[OPK05a] R. Obermaisser, P. Peti, and H. Kopetz. Virtual gateways in the DECOS integrated architecture. In *Proceedings of the Workshop on Parallel and Distributed Real-Time Systems 2005 (WPDRTS)*. IEEE, April 2005.

[OPK05b] R. Obermaisser, P. Peti, and H. Kopetz. Virtual networks in an integrated time-triggered architecture. In *Proceedings of the 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS2005)*, pages 241–253, Sedona, Arizona, February 2005.

[OSE03] OSEK/VDX. *Network Management, Version 2.5.2*, January 2003.

[PABD+99] D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabejac, and A. Wellings. GUARDS: a generic upgradable architecture for real-time dependable systems. *IEEE Transactions on Parallel and Distributed Systems*, 10:580–599, June 1999.

[Par01] M. Parsons. Design and manufacture of automotive pressure sensors. *Sensors*, April 2001.

[Pau05] H. Paulitsch. Fault injection for diagnosis and maintenance in the time-triggered architecture. Master thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, September 2005.

[PBC+02] C. Picardi, R. Bray, F. Cascio, L. Console, P. Dague, O. Dressler, D. Millet, B. Rehfus, P. Struss, and C. Vallee. IDD: integrating diagnosis in the design of automotive systems. In *Proceedings of the Fifteenth European Conference on Artificial Intelligence, ECAI 2002*, Lyon, France, July 2002.

[PDN+01] M. Pecht, M. Dube, M. Natishan, R. Williams, J. Banner, and I. Knowles. Evaluation of built-in test. *IEEE Transactions on Aerospace and Electronic Systems*, 37(1):266–271, January 2001.

[PDT03] K.W. Przytula, D. Dash, and D. Thompson. Evaluation of bayesian networks used for diagnostics. In *Proceedings of the IEEE Aerospace Conference*, volume 7, pages 3177–3187, March 2003.

[Pec01] M. Pecht. Electronic reliability engineering in the 21st century. In *Proceedings of 2001 International Symposium on Electronic Materials and Packaging*, pages 1–7. IEEE, 2001.

[PK03] Z. Peng and N. Kessissoglou. An integrated approach to fault diagnosis of machinery using wear debris and vibration analysis. *Wear*, 255(7–12):1221–1232, August 2003.

[PKOP99] M.J. Pont, R. Kureemun, H.L.R. Ong, and W. Peasgood. Increasing the reliability of embedded automotive applications in the presence of EMI: a pilot study. In *Proceedings of the IEE Seminar on Electromagnetic Compatibility for Automotive Electronics*, pages 4/1–4/4, September 1999.

[PL00] P. Pettersson and K.G. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, February 2000.

[PM98] B. Pauli and A. Meyna. Ein praxisorientierter Ansatz zur Bestimmung von komulierten und durchschnittlichen Ausfallraten. *Automobiltechnische Zeitschrift (ATZ)*, pages 382–386, 1998.

[PMH98] B. Pauli, A. Meyna, and P. Heitmann. Reliability of electronic components and control units in motor vehicle applications. In *VDI Berichte 1415, Electronic Systems for Vehicles*, pages 1009–1024. Verein Deutscher Ingenieure, 1998.

[Pod90] A.S. Podgorski. Lightning standards for aircraft protection. In *Proceedings of the IEEE International Symposium on Electromagnetic Compatibility*, pages 218–223, August 1990.

[Pol95a] S. Poledna. Fault tolerance in safety critical automotive applications: cost of agreement as a limiting factor. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*, pages 73–82, June 1995.

[Pol95b] S. Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, November 1995.

[Pol04] S. Poledna. TTP-Tools – The tool set of the Time-Triggered Architecture. In *Proceedings of the Monterey Workshop*, Baden, Austria, October 2004.

[POT+05] P. Peti, R. Obermaisser, F. Tagliabo, A. Marino, and S. Cerchio. An integrated architecture for future car generations. In *Proceedings of the 8th IEEE International Symposium on Object-oriented Real-time distributed Computing*, May 2005.

[Pow92] D. Powell. Failure mode assumptions and assumption coverage. In *Proceedings of the 22nd IEEE Annual International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 386–395, Boston, USA, July 1992.

[PP01] V. Polimac and J. Polimac. Assessment of present maintenance practices and future trends. In *Proceedings of the Transmission and Distribution Conference and Exposition, IEEE/PES*, pages 891–894, 2001.

[PR92] M. Pecht and V. Ramappan. Are components still the major problem: a review of electronic system and device field failure returns. *IEEE Transactions on Components, Hybrids, and Manufacturing Technology*, 15(6):1160–1164, December 1992.

[PW03] B. Peischl and F. Wotawa. Model-based diagnosis or reasoning from first principles. *IEEE Intelligent Systems*, 18(3):32–37, May 2003.

[Ram92] G. Ramohalli. The Honeywell on-board diagnostic and maintenance system for the Boeing 777. In *Proceedings of the 11th IEEE/AIAA Digital Avionics Systems Conference*, pages 485–490, October 1992.

[Ram01] A. Ramakrishnan. *The Avionics Handbook*, chapter Electronic Hardware Reliability. CRC Press LCC, 2001.

[Ran01] H. Ranter. Access to air safety information. In *Proceedings of the 8th Annual MRO Regulations, Quality & Safety Conference*, October 2001.

[Rei03] H.-R. Reichel. *Elektronische Bremssysteme*. expert Verlag, Renningen, 2nd edition, 2003.

[RH04] G. Reichart and M. Haneberg. Key drivers for a future system architecture in vehicles. In *Convergence International Congress*, Detroit, MI, USA, October 2004. SAE.

[RLT78] B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys*, 10(2):123–165, 1978.

[Rom00] D. Romanchik. Auto electronics reliability: It ain't what it used to be. *Automotive Test Report*, February 2000.

[Ros92] D.S. Rosenblum. Towards a method of programming with assertions. In *Proceedings of the 14th international conference on Software engineering*, pages 92–104. ACM Press, 1992.

[RTA00] RTAI Programming Guide, Version 1.0. Dipartimento di Ingegneria Aerospaziale Politecnico di Milano (DIAPM), Italy, September 2000. Available at `http://www.rtai.org`.

[RTC92] Radio Technical Commission for Aeronautics, Inc. (RTCA), Washington, DC. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, December 1992.

[RTC99] Radio Technical Commission for Aeronautics, Inc. (RTCA). *RTCA-SC-182/EUROCAE WG-48: The Minimum Operational Performance Standards for Avionics Computer Resource (ACR)*, June 1999.

[Rus36] B. Russell. *Proc. Camb. Philos. Soc.*, 32:216–228, 1936.

[Rus99] J. Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Also to be issued by the FAA.

[Rus01a] J. Rushby. Bus architectures for safety-critical embedded systems. In Tom Henzinger and Christoph Kirsch, editors, *Proceedings of the First Workshop on Embedded Software (EMSOFT 2001)*, volume 2211 of *Lecture Notes in Computer Science*, pages 306–323, Lake Tahoe, CA, October 2001. Springer-Verlag.

[Rus01b] J. Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, September 2001.

[Rus01c] J. Rushby. Modular certification. Technical report, Computer Science Laboratory SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, USA, September 2001.

[RX97] A. Ran and J. Xu. Architecting software with interface objects. In *Proceedings of the 8th Israeli Conference on Computer Systems and Software Engineering*, pages 30–37, Herzliya, Israel, June 1997. Software Technol. Lab., Nokia Res. Center, Espoo.

[SC97] M. Sanseverino and F. Cascio. Model-based diagnosis for automotive repair. *IEEE Expert*, pages 33–37, November 1997.

[Sch90] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[Sch95] A. Schedl. The short-term stability of crystal oscillators: Experimental results. Research Report 1/1995, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 1995.

[Sch96] A. Schedl. *Design and Simulation of Clock Synchronization in Distributed Systems.* PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 1996.

[Scu98] J.K. Scully. The hidden crisis in test effectiveness. In *Proceedings of the AUTOTESTCON 1998*, pages 59–66, August 1998.

[Sim96] H.A. Simon. *The Sciences of the Artificial*. MIT Press, 1996.

[SJK00] C. Skaanning, F.V. Jensen, and U. Kjaerulff. Printer troubleshooting using bayesion networks. In R. Loganantharaj, editor, *Lecture Notes in AI*, volume 1821, pages 367–380. Spinger-Verlag, 2000.

[SKSS94] B.A. Sorensen, G. Kelly, A. Sajecki, and P.W. Sorensen. An analyzer for detecting intermittent faults in electronic devices. In *Proceedings of the IEEE Systems Readiness Technology Conference (AUTOTESTCON 94)*, pages 417–421, 1994.

[SM99] J. Swingler and J.W. McBride. The degradation of road tested automotive connectors. In *Proceedings of the 45th IEEE Holm Conference on Electrical Contacts*, pages 146–152, Pittsburgh, PA, USA, October 1999. Dept. of Mech. Eng., Southampton Univ.

[SMM00] J. Swingler, J.W. McBride, and C. Maul. Degradation of road tested automotive connectors. *IEEE Transactions on Components and Packaging Technologies*, 23(1):157–164, March 2000.

[SS01] S. Sullivan and G. Slenski. Managing electrical connection systems and wire integrity on legacy aerospace vehicles. In *Proceedings of the FAA Principal Inspectors and Engineers Workshop*, Seattle, USA, 2001.

[SSW00] M. Sachenbacher, P. Struss, and R. Weber. Advances in design and implementation of OBD functions for diesel injection based on a qualitative approach to diagnosis. In *Proceedings of SAE 2000 World Congress*, Detroit, MI, USA, 2000. SAE.

[Sta97] A.G. Starr. A structured approach to the selection of condition based maintenance. In *Proceedings of the Fifth International Conference on Factory 2000 - The Technology Exploitation Process*, pages 131–138, 1997.

[SV02] A. Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign of EETimes*, February 2002.

[SW04] A. Saad and U. Weinmann. Intelligent automotive system services: Requirements, architectures and implementation issues. In *Convergence International Congress*, Detroit, MI, USA, October 2004. SAE.

[SWH95] N. Suri, C.J. Walter, and M.M. Hugue. *Advances In Ultra-Dependable Distributed Systems*, chapter 1. IEEE Computer Society Press, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA 90720-1264, 1995.

[Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[Tan93] P.N. Tanner. Automotive connectors. In *Proceedings of the IEE Colloquium on Connectors on Vehicles*, pages 6/1–6/10, November 1993.

[TAP02] D.A. Thomas, K. Ayers, and M. Pecht. The 'trouble not identified' phenomenon in automotive electronics. *Microelectronics Reliability*, 42:641–651, 2002.

[TE01] R. Tappe and D. Ehrhardt. Dynamic tests in complex systems. In *Proceedings of the International Test Conference*, pages 609–614. IEEE, 2001.

[TS01] C. Teal and D. Sorensen. Condition based maintenance [aircraft wiring]. In *Proceedings of the 20th Conference on Digital Avionics Systems, DASC*, volume 1, pages 3B2/1–3B2/7, October 2001.

[TTP05] TTTech Computertechnik AG, Schönbrunner Strasse 7, A-1040 Vienna, Austria. *TTPOS: The OSEKtime-Based Operating System for Safety-Critical Real-Time Applications*, 2005.

[TTT02a] TTTech Computertechnik AG, Schönbrunner Strasse 7, A-1040 Vienna, Austria. *TTP Monitoring Node – A TTP Development Board for the Time-Triggered Architecture*, March 2002.

[TTT02b] TTTech Computertechnik AG, Schönbrunner Strasse 7, A-1040 Vienna, Austria. *TTP/C Controller C2 Controller-Host Interface Description Document, Protocol Version 2.1*, November 2002.

[TTT04a] TTTech Computertechnik AG, Schönbrunner Strasse 7, A-1040 Vienna, Austria. *TTP Development Cluster - The Complete TTP System*, 2004.

[TTT04b] TTTech Computertechnik AG, Schönbrunner Strasse 7, A-1040 Vienna, Austria. *TTP Powernode – The TTP Development Board*, 2004.

[TTT05] TTTech Computertechnik AG, Schönbrunner Strasse 7, A-1040 Vienna, Austria. *TTP Disturbance Node: Testing Fault Tolerance Properties of Safety-Critical Distributed Real-Time Systems*, 2005.

[Uni03] Universal Synaptics Corporation, 1801 West 21st Street, Ogden, Utah, USA. *Wiring Problems, No Faults Found, Anomalies and Politics*, 2003.

[vB83] J. van Benthem. *The Logic of Time*. D. Reidl Publishing Company, 1983.

[VB94] S.L.E. Varner and T.D. Belle. Maintenance terminal function: the user interface for the boeing 777 onboard maintenance system. In *Proceedings of the 13th AIAA/IEEE Digital Avionics Systems Conference (DASC)*, pages 6–11, October 1994.

[vDvM96] P. van Dijk and F. van Meijl. Contact problems due to fretting and their solutions. *AMP Journal of Technology*, 5:14–18, June 1996.

[VM94] J.M. Voas and K.W. Miller. Putting assertions in their place. In *Proceedings of the 5th International Symposium on Software Reliability Engineering*, pages 152–157. IEEE, November 1994.

[Wal02] W. Waldeck. Diagnostic protocol challenges in a global environment. In *Convergence International Congress & Exposition On Transportation Electronics*, Detroit, MI, USA, October 2002. SAE.

[Was95] F. Waskiewicz. An object-oriented framework for manufacturing applications. OMG BOMSIG, Ottowa, 1995.

[WBC00] W. Wondrak, A. Boos, and R. Constapel. Design for reliability in automotive electronics. In *Proceedings of the Microtec 2000*, Hannover, EXPO, July 2000.

[WCRV00] J.M. Wetzer, G.J. Cliteur, W.R. Rutgers, and H.F.A. Verhaart. Diagnostic- and condition assessment-techniques for condition based maintenance. In *Proceedings of the 2000 Annual Report Conference on Electrical Insulation and Dielectric Phenomena*, volume 1, pages 47–51, 2000.

[Web92] J.P. Weber. Integrated diagnostics for software. In *Proceedings of the IEEE 1992 National Aerospace and Electronics Conference*, volume 2, pages 565–571, May 1992.

[Whi90] G.J. Whitrow. *The Natural Philosophy of Time*. Oxford Univeristy Press, second edition, 1990.

[Wie14] N. Wiener. A contribution to the theory of relative position. *Proc. Camb. Philos. Soc.*, 17:441–449, 1914.

[Wij01] J.G. Wijnstra. Components, Interfaces and Information Models within a Platform Architecture. In Jan Bosch, editor, *Generative and Component-Based Software Engineering*, LNCS 2186, pages 25–35, Erfurt, Germany, September 2001. Springer.

[WK94] S. Williams and C. Kindel. The component object model: A technical overview. Microsoft Corporation, White Paper, October 1994.

[WLS97] C.J. Walter, P. Lincoln, and N. Suri. Formally verified on-line diagnosis. *IEEE Transactions on Software Engineering*, 23(11):684–721, November 1997.

[Won99] W. Wondrak. Physical limits and lifetime limitations of semiconductor devices at high temperatures. *Microelectronics Reliability*, 39:1113–1120, 1999.

[WP99] D. Wybo and D. Putti. A qualitative analysis of automatic code generation tools for automotive powertrain applications. In *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design*, pages 225–230, Hawaii, USA, August 1999.

[WWS99] J. Wilde, W. Wondrak, and W. Senske. Reliability requirements for microtechnologies used in automotive applications. In *Proceedings of the Congress for Microsystems and Precision Engineering, MicroEngineering 99*, Stuttgart, Germany, October 1999. Stuttgarter Messe- und Kongressgesellschaft GmbH.

[Yeh98] Y.C. Yeh. Design considerations in Boeing 777 fly-by-wire computers. In *Proceedings of the 3rd IEEE International High-Assurance Systems Engineering Symposium*, pages 64–72, November 1998.

[Zad69] L.A. Zadeh. *The concept of system, aggregate, and state in system theory*, volume 8 of *Inter-University Electronics Series*, pages 3–42. McGraw-Hill, 1969.

Curriculum Vitae

Philipp Peti

November 5^{th} 1977	Born in Vienna, Austria
September 1984 – June 1988	Elementary School in Vienna
September 1988 – June 1996	Secondary School in Vienna
October 1996 – May 2001	Studies of Computer Science at the Vienna University of Technology
May 2001	Master's Degree in Computer Science
June 2001 – June 2002	Civilian Service
August 2002 – October 2002	Project Assistant at the Vienna University of Technology
since October 2002	PhD Studies and Research/Teaching Assistant at the Vienna University of Technology