07ATC-203

The Fault Assumptions in Distributed Integrated Architectures

R. Obermaisser
Vienna University of Technology, Austria

P. Peti
Vienna University of Technology, Austria

ABSTRACT

Distributed integrated architectures in the automotive and avionic domain result in hardware cost reduction, dependability improvements, and improved coordination between application subsystems compared to federated systems. In order to support safety-critical application subsystems, a distributed integrated architecture needs to support fault-tolerance strategies that enable the continued operation of the system in the presence of failures. The basis for the implementation and validation of fault-tolerance strategies are realistic fault assumptions, which are captured in a fault hypothesis. This paper describes a fault hypothesis for distributed integrated architectures, which takes into account the sharing of the communication and computational resources of a single distributed computer system among multiple application subsystems. Each node computer serves for the execution of multiple jobs. In analogy, the communication network interconnecting the node computers has to support message exchanges of more than one application subsystem. Using a generic system model of a distributed integrated architecture, we argue in favor of a differentiation of fault containment regions for hardware and software faults. Based on these fault containment regions, we discuss the failure modes, the failure rates, the maximum number of failures, and the recovery intervals. In particular, the fault hypothesis describes the assumptions concerning the respective frequencies of transient and permanent failures in consideration of recent semiconductor trends.

1. INTRODUCTION

Computer systems of X-by-wire applications in the automotive and avionic domain rank among the computer systems with the highest dependability requirements. In general, one demands a level of safety that is higher or equal to the level of safety of the systems that are replaced. In these ultra-dependable applications, a maximum failure rate of 10^{-9} critical failures per hour is demanded [1][p. 10]. This can only be achieved by utilizing fault-tolerant strategies that enable the continued operation of the system in the presence of component failures [2].

Since systems cannot be tested to a reliability in the order of 10^{-9} failures/hour, a combination of experimental evidence and formal reasoning using a reliability model is needed to construct the safety argument [3]. The fault hypothesis [4] of a distributed system is the basis not only for the design and validation of the fault-tolerance algorithms, but also for the evaluation of the assumption coverage [5]. For this reason, the fault hypothesis plays a major role in the safety argument of a safety-critical distributed real-time system.

The objective of this paper is the definition of a fault hypothesis for integrated architectures, such as the *DECOS integrated architecture* [6], Integrated Modular Avionics (IMA) [7], or Automotive Open System Architecture (AUTOSAR) [8]. We assume a shared distributed computer system, where the node computers are interconnected by a time-triggered network. Each node computer is shared among multiple jobs in order to overcome the "1 Function – 1 Electronic Control Unit (ECU)" limitation [9, 10]. Also, a single time-triggered physical network handles the message exchanges between the node computers hosting the application subsystems. Each application subsystem is provided with guaranteed communication resources. In analogy, guaranteed computational resources of the node computers (e.g., CPU time, memory, I/O) are assigned to jobs by employing a *partition management operating system*.

In an integrated architecture, the sharing of the node computers and the common physical network among jobs from different application subsystems determines the assumptions concerning fault containment regions in the fault hypothesis. The basic idea is the differentiation of fault containment regions for software and for hardware faults. This paper extends the existing fault hypothesis of the Time-Triggered Architecture (TTA) [4], which regards each node computer as an atomic unit in the fault hypothesis. This node-centric view is characteristic for federated systems and encompasses no discrimination between hardware and software faults.

In the introduced fault hypothesis, we recognize that the different jobs on a node computer are to a high degree independent with respect to software faults. Firstly, the different jobs provide different application services. Furthermore, the jobs integrated within a node computer typically originate from different vendors, each employing its own design teams, development tools, and development processes. This insight is significant for supporting mixed criticality systems, in which jobs with different criticality levels are collocated on shared node computers. Based on the assumptions in the fault hypothesis, the architecture must support error containment between jobs in order to enable modular certification [11] of the complete computer system. Otherwise an elevation of the criticality for all jobs to the highest criticality level of a software module in the system would become necessary.

The paper is structured as follows. Section 2 presents a generic system model for integrated system architectures. Section 3 provides examples of integrated architectures that are compatible with the presented system model and the introduced fault hypothesis. We describe the general structure of a fault hypothesis for safety-critical distributed real-time systems in Section 4. Based on this general structure of a fault hypothesis, Section 5 provides an instantiation for integrated architectures. The paper finishes with a conclusion in Section 6.

2. SYSTEM MODEL OF AN INTEGRATED ARCHITECTURE

In integrated architectures computational resources (e.g., processor time, memory) and communication resources (i.e., network) are shared among multiple jobs in order to reduce the number of deployed node computers, associated wiring, and to avoid unnecessary resource duplication (e.g., redundant sensors). In the following we discuss our system model of an integrated architecture together with the resulting communication and component model.

SYSTEM MODEL

A standard problem solving technique is the division of complex systems into nearly-independent subsystems [12]. 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. The major challenge in this context is the problem of composability and compositionality, ensuring that the pre-validated properties hold also after system integration [13, 14].

Today large distributed computer systems are typically constructed out of node computers (e.g., in the automotive domain). Such node computers are delivered by a vendor according to the requirements document stating which services must be provided at the interface of the node computer. The node computers provided by the vendor are subject to comprehensive tests by the Original Equipment Manufacturer (OEM) to determine correctness. For this task, Hardware in the Loop (HiL) simulation is of widespread use to analyze the interaction of node computers in the target system [15].

However, there is a strong tendency having vendors supply jobs (i.e., software components) instead of node computers. In an additional step these jobs are then allocated to the node computers of the target platform [16, 17]. This strongly relates to the field of Component-Based Software Engineering (CBSE) [18, 19], which is concerned with the rapid assembly of software systems from pre-built components of independent vendors [20].

Therefore, a system model as depicted in Figure 1 will replace the prevalent “1 Function – 1 ECU” design methodology. In contrast to these federated architectures, integrated architectures allow the deployment of multiple jobs on one node computer. In general, the overall functionality of such a system is composed of several application subsystems (e.g., steer-by-wire subsystem [21]), each designed like a classic federated system to allow independent development and validation [22].

In order to provide an execution environment that allows execution of jobs without unintended interference, temporal and spatial partitioning for both computational and communication resources is required:

- *Partitioning of Computational Resources.* The purpose of spatial partitioning is to prevent jobs from overwriting memory elements of other jobs (i.e., protection of data and code) and to prevent jobs in interfering in the access of devices [23]. A common way for establishing spatial partitioning is the use of hardware memory protection mechanisms, e.g., a Memory Management Unit (MMU) [24]. The MMU tables, managed by the operating system, provide means for spatial partitioning by determining the areas of physical memory that can be accessed by a single job.

  The purpose of temporal partitioning is the retention of a correct schedule even in the case of faulty jobs holding the shared resource. Especially, in the context of real-time systems it is thus required, that no job can delay other jobs by holding a shared resource (e.g., the processor). For more information

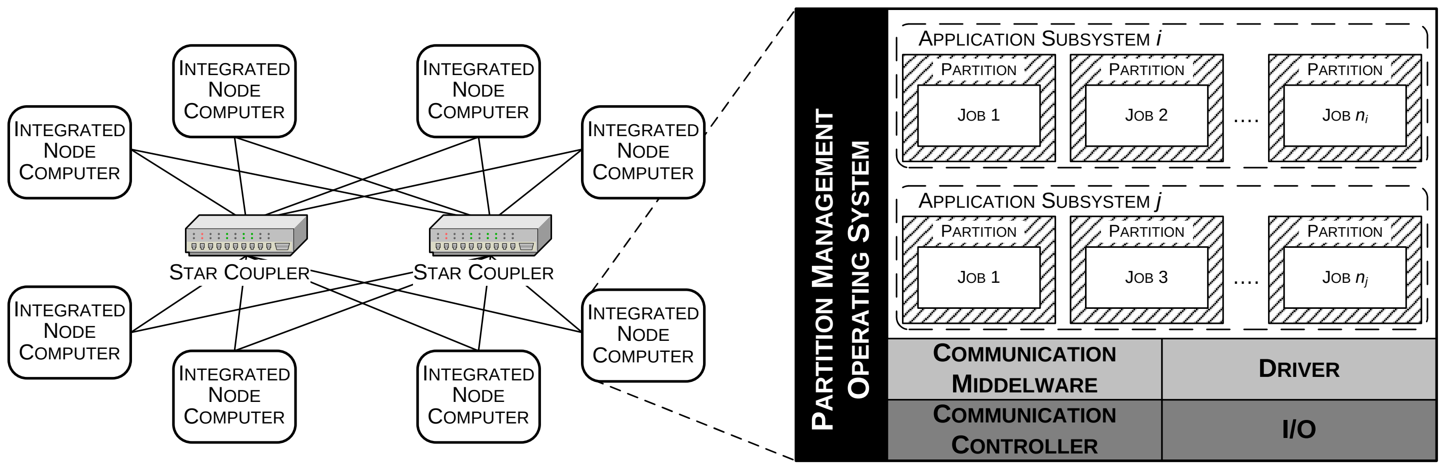

Figure 1: Integrated Architecture

see also [25].

- *Partitioning of Communication Resources.* Spatial partitioning in the context of the communication resources addresses the problem of faulty jobs manipulating messages of other jobs via unintended access (e.g., altering of messages in message buffers not owned by the job). Temporal partitioning addresses the issue that no job shall be able to affect the bandwidths and latencies of the communication activities of other jobs.

MODEL OF THE COMMUNICATION SYSTEM

In integrated systems one has to distinguish between the communication service interconnecting the integrated node computers and the communication service to exchange messages between the jobs.

In our model, the communication system between the node computers is based on a time-triggered protocol such as TTP [26], FlexRay [27] or SafeBus [28]. The rationale behind choosing a time-triggered communication protocol is the suitability for ultra-dependable systems [29]. Time-triggered communication protocols are characterized by a guaranteed message transport with low jitter, error containment between node computers, and a fault-tolerant distributed global clock service.

On top of the time-triggered network, for each application system a dedicated part of the available communication slots is reserved according to the requirements of the application. This way temporal properties (e.g., latencies, bandwidth) for the message exchanges between jobs can be guaranteed. Architectural services are required to allow a mapping of the underlying time-triggered communication system.

MODEL OF AN INTEGRATED NODE COMPUTER

An integrated node computer provides an execution environment for multiple collocated jobs of one or more application subsystems as shown in Figure 1. The model of an integrated node computer comprises:

- *Jobs:* The jobs implement the application functionality. A job is part of an application subsystem and represents the unit of distribution. Each job is the responsibility of a single organizational entity (e.g., a specific supplier). The interaction with other jobs occurs through the communication services provided by the communication middleware.

- *Partition management operating system.* The purpose of the partition management operating system is the establishment of multiple encapsulated execution environments for combining multiple jobs within a single node computer. The encapsulated execution environment provided for a job is denoted as a *partition* and provides guaranteed computational resources (CPU time, memory). The partition management operating system implements mechanisms for spatial and temporal partitioning in order to protect the computational resources of the individual partitions. The scheduling of partitions needs to ensure that a timing failure of a job, such as a worst-case execution time violation, does not affect the CPU time available to other partitions. In analogy, the spatial partitioning mechanisms of the partition management operating system include memory protection between partitions (e.g., hardware-enforced with a MMU). Thereby, each partition emulates a virtual node computer that is dedicated to a single job only.

  An example for a partition management operating system is Wind River's VxWorks AE653 [30]. In compliance with ARINC standard 653 [31] (also called APplication EXecutive (APEX)), this real-time operating system supports the execution of avionic jobs that would be implemented on dedicated Line Replaceable Units (LRUs) in non IMA systems. VxWorks AE653 supports an additional operating system running within a partition, e.g., replacing the OS that has been deployed with a legacy application in the respective LRU. Other examples for partition management operating systems include LynxOS-178 [32] and a DECOS operating system based on Linux Real-Time Application Interface (RTAI) [25].

- *Communication Middleware:* The main purpose of the middleware is the management of the communication resources as previously described. The middleware provides a technology invariant interface to the jobs that abstracts from any hardware-specific implementation details. For example, in AUTOSAR [17] the runtime environment (RTE) provides such a generic communication service for the applications. In the DECOS the high-level virtual network services perform this task [33].

- *Communication controller:* The purpose of the communication controller is to provide access to the underlying time-triggered communication system. By the use of hardware drivers and the provision of standardized Application Programming Interface (API) one typically abstracts from the used hardware and thus ensures reuse of existing code in future systems.

- *Input/Output (I/O) and Drivers:* The jobs hosted on a node computer exploit the input/output subsystem for interacting with the controlled object and the human operator. This interaction occurs either via a direct connection to sensors and actuators or via a fieldbus (e.g., Local Interconnect Network (LIN) [34]). 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.

  In a federated system, each job has full access to I/O of the respective node computer. On an integrated node computer, an I/O partition is required that manages I/O from the various jobs.

3. EXAMPLES OF INTEGRATED ARCHITECTURES

This section gives an overview of exemplary integrated architectures (IMA, AUTOSAR, DECOS) that are compatible with the presented system model and the introduced fault hypothesis.

AUTOSAR

The AUTOSAR [8] is a system architecture and development methodology for automotive electronic systems. Among the primary goals of AUTOSAR are the standardization of the basic software of a node computer (called ECU in AUTOSAR) and the ability to integrate jobs (called software components in AUTOSAR) from multiple suppliers. Another focus of AUTOSAR is the establishment of portability and location transparency for software components in order to accommodate future changes to the automotive electronic systems and facilitate reuse across product lines [35].

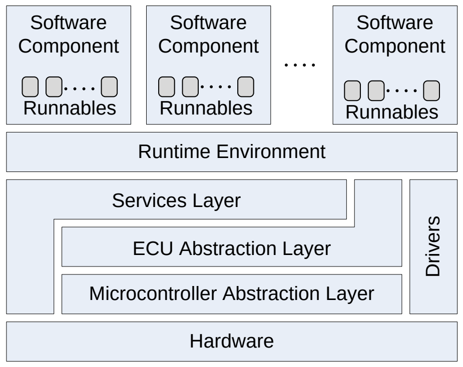

Figure 2: Node Computer in AUTOSAR

Model of a Node Computer. A node computer in AUTOSAR supports the integration of software components from multiple suppliers (see Figure 2). A *software component* is a piece of software that can be mapped to a node computer. Internally, a software component contains threads of control called *runnables*. AUTOSAR distinguishes between event-triggered and communication-triggered runnables. The execution of event-triggered runnables is triggered by the occurrence of a significant event (e.g., timeout). Communication-triggered runnables are activated by the arrival of messages from other software components.

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

Below the runtime environment in Figure 2, the node computer in AUTOSAR contains the basic software encompassing layers with standardized functionality. These layers consist of software components that do not fulfill application functionality themselves, but interact with the application software components using the runtime environment.

- **Services layer.** This layer provides operating system services (e.g., priority-based scheduling of runnables, memory management), and communication services (e.g., communication stack for FlexRay).

- **ECU abstraction layer.** For accessing peripherals and devices, this layer establishes an API that abstracts from the underlying hardware. For example, the I/O hardware and the communication hardware (e.g., FlexRay controller) deployed on the node computer are hidden from the upper layers.

- **Microcontroller abstraction layer.** This layer provides access to internal and external peripherals of the node computer.

- **Drivers.** The drivers are application-dependent and interact with sensors and actuators. In order to satisfy tight timing requirements (e.g., injection control), the drivers can use interrupts and directly access the microcontroller and the peripherals.

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

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

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

INTEGRATED MODULAR AVIONICS

While present day automotive electronic systems adhere to the federated architectural paradigm, integrated architectures have already been successfully applied in the development of the electronic systems of commercial aircrafts (e.g., 777 [36]). ARINC standard 651 [7] is known as IMA and addresses the design of integrated avionic systems. The construction of an IMA architecture relies on several other ARINC standards. For instance, the services of the avionic software environment are specified by ARINC 653 [37], which is known as APEX. APEX provides services for partition management, process management, time management, memory management, interpartition communication, intrapartition communication, and diagnosis. IMA systems are not restricted to a particular communication network. Common standards for the communication network include point-to-point protocols such as ARINC 429 [38], event-triggered communication systems (e.g., ARINC 664 [39], ARINC 629 [40]), and time-triggered communication systems (e.g., ARINC 659 (SAFEbus) [41]).

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

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

APEX distinguishes four types of software modules. The application partitions and the system partitions form the application software. The core software consists of the O/S kernel and system-specific functions.

- **Application partitions** execute software implementing application functionality (i.e., the jobs). For error containment reasons, application partitions may only use ARINC 653 calls to interface the hardware and communication system.
- **System partitions** require interfaces outside of APEX services. System partitions are specific to the core software implementation.
- **The O/S kernel** provides the services defined by the APEX specification.
- **System-specific functions** implement device drivers, diagnostic, and maintenance functions.

Tasks within partitions can employ intrapartition communication mechanisms, thereby avoiding the runtime overhead of message passing. Intrapartition communication mechanisms include buffers, blackboards, semaphores, and events. Buffers and blackboards allow general communication and synchronization between tasks in a partition, while semaphores and events are used for synchronization only.

In order to protect the memory of partitions and to avoid interference in the temporal domain (e.g., through a task overrunning its deadline or blocking a shared resource), APEX demands sufficient processing, I/O and memory resources from the used processor. Furthermore, APEX requires time resources, atomic operations, and mechanisms for transferring control to the operating system, if a partition attempts to perform an invalid operation.

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

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

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

DECOS

The DECOS integrated architecture [6] aims at providing 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. Generic architectural services separate the application functionality from the underlying platform technology in order to facilitate reuse and reduce design complexity.

In order to maximize the number of platforms and applications that can be covered, the DECOS architecture 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.

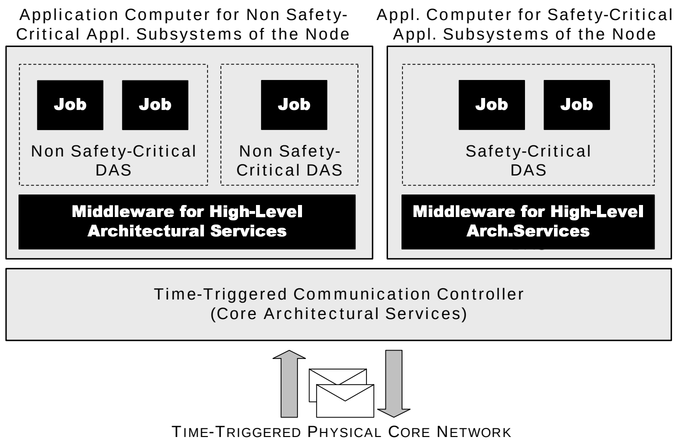
Application Computer for Non Safety-Critical Appl. Subsystems of the Node
Appl. Computer for Safety-Critical Appl. Subsystems of the Node
Job
Job
Job
Job
Job
Non Safety-Critical DAS
Non Safety-Critical DAS
Safety-Critical DAS
Middleware for High-Level Architectural Services
Middleware for High-Level Arch.Services
Time-Triggered Communication Controller (Core Architectural Services)
TIME-TRIGGERED PHYSICAL CORE NETWORK

Figure 3: Example of a Node Computer in DECOS

Based on the core services, the DECOS integrated architecture realizes high-level architectural services, which constitute the interface for the jobs to the underlying platform. Among the high-level services are virtual network services [33], diagnostic services [42], and gateway services [43].

Model of a Node Computer. In a node computer (see Figure 3) of the DECOS architecture one can distinguish between several layers, namely the realization of the core architectural services, the realization of the high-level architectural services and the application layer. The application layer is comprised by one or more application computers hosting the jobs. Each job is executed in a dedicated partition and communicates with other jobs of the same application subsystem by utilizing the virtual network services [33].

Communication System. In the DECOS architecture, each application subsystem is provided with a dedicated communication infrastructure that is realized as a *virtual network*. A virtual network is established as an overlay network on top of a time-triggered physical network [33]. The time-triggered physical network has to support clock synchronization and the periodic exchange of state messages at a priori specified global points in time. Any communication protocol that supports these capabilities can be used as a basis for the construction of virtual networks. Examples of suitable protocols are the Time-Triggered Protocol (TTP) [44], and FlexRay [45].

A job can send two types of messages on a virtual network, namely *state messages* and *event messages*. These two message types differ w.r.t. the information semantics [1, p. 31]. While state messages contain the absolute value of a real-time entity (e.g., speed is $10\,\mathrm{ms}^{-1}$),

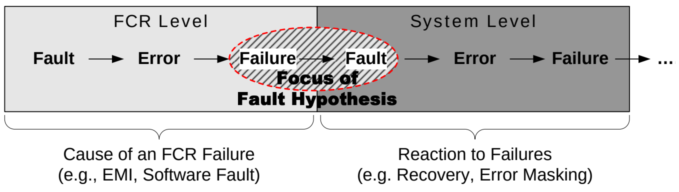
FCR Level
System Level
Fault
Error
Failure
Fault
Error
Failure
...
Focus of Fault Hypothesis
Cause of an FCR Failure (e.g., EMI, Software Fault)
Reaction to Failures (e.g. Recovery, Error Masking)

Figure 4: Relationship between Fault-Error-Failure Chain and Fault Hypothesis

event messages transport relative values (e.g., decrease of speed by $2\,\mathrm{ms}^{-1}$). The message type determines the interface (i.e., the ports) between the job and the virtual network.

4. CONSTITUTING PARTS OF A FAULT HYPOTHESIS

The *fault hypothesis* specifies assumptions about the types of faults, the rate at which components fail and how components may fail [5]. The *assumption coverage* is the probability that these assumptions hold in reality. Since fault-tolerance mechanisms of a system are based on these assumptions, the complete system may fail in case the assumptions concerning faults, failure rates, and failure modes are violated.

FAULT-CONTAINMENT REGIONS

A *Fault Containment Region (FCR)* is defined as a subsystem that operates correctly regardless of any arbitrary logical or electrical fault outside the region [46]. The justification for building ultra-reliable systems from replicated resources rests on an assumption of failure independence among redundant units [2]. The independence of FCRs can be compromised by shared physical resources (e.g., power supply, timing source), external faults (e.g., Electromagnetic Interference (EMI), spatial proximity) and design.

For example, fault injection experiments in the TTA have demonstrated that the failure of a local bus guardian is correlated with a failure of the respective node computer due to shared power supply, timing source and physical proximity [47]. Therefore, the TTA [48] has been adapted to perform timing failure detection by two replicated central guardians [49].

In conformance with the fault-error-failure chain introduced by Laprie [50], one can distinguish between faults that cause the failure of a FCR (e.g., design of the hardware or software of the FCR, operational fault of the FCR) and faults at the system-level. The latter type of fault is a failure of a FCR, which needs to be tolerated by the distributed fault-tolerance mechanisms (see Figure 4). For example, a common approach for masking component failures is N-modular redundancy (NMR) [51, 52, 53, 54]. N replicas receive the same requests and provide the same service. The output of all replicas is provided to a voting mechanism, which selects one of the results (e.g., based on majority) or transforms the results to a single one (average voter). The most frequently used N-modular configuration is triple-modular redundancy (TMR).

Since the FCR failures are relevant for the design of the fault tolerance mechanisms, they are the content of the fault hypothesis. A system-level failure can result from the inability to handle such a system-level fault.

FAILURE MODE ASSUMPTION

Failure modes of FCRs are defined through the effects as perceived by the service user, i.e., independently of the actual cause or rate of failures. A formal definition in terms of assertions on the sequences of value-time tuples can be found in [5]. Failure modes determine the degree of redundancy required to ensure correct error processing. Based on the rigidity of assumptions, the following hierarchy of failures modes can be established [55]:

- **Fail-stop failures:** A fail-stop failure is defined as a FCR behavior, where the FCR does not produce any outputs. The FCR omits to produce output to subsequent inputs until it restarts. It is additionally assumed that all correct FCRs detect the fail-stop failure.
- **Crash Failures:** A FCR suffering a crash failure does not produce any outputs. In contrast to fail-stop failures, a crash failure can remain undetected for correct FCRs.
- **Omission Failures:** An omission failure occurs, if the sender FCR fails to send a message, or the receiver fails to receive a sent message. As a consequence, the receiver does not respond to an input. The detection of an omission failure is not guaranteed.
- **Timing Failures:** The FCR does not meets its temporal specification. Outputs of a FCR are delivered too early or too late.
- **Byzantine or Arbitrary Failures:** There is no restriction on the effects a service user may perceive. Arbitrary failures include the forging of messages and "two-faced" FCR behaviors [56].

A different classification of failure modes can be found in [57] and distinguishes the following additional types of failure modes:

- **Babbling Idiot.** In case of a babbling idiot failure, the FCR does not obey its temporal specification by sending untimely messages. For example, in a CAN network [58] a FCR (i.e., a CAN node) constantly sends high-priority messages thus monopolizing the network.

- **Slightly-off-Specification (SoS).** Such a failure is a special type of Byzantine failure. One can distinguish temporal and value SoS failures. An example for a value SoS failure is an intermediate electrical voltage that is close to the threshold between logical 0 and logical 1 and can be perceived with different logical values by different observers [59]. An example for a temporal SoS failure is a message with a receive instant that is slightly outside the boundary of the interval of correct receive instants. In such a case, due to the inability to perfectly synchronize clocks one node can classify the message as timely, whereas another node may detect a message timing failure.

- **Masquerading.** Masquerading is defined as the sending or receiving of messages using the identity of another principal without authority [60, p. 480].

FAILURE RATES ASSUMPTIONS

Part of the fault hypothesis is a specification of the failure rate of FCRs. In general, a differentiation of failure rate with respect to different failure modes and the failure persistence is necessary. For example, fault injection experiments [61] have shown that restrictive failure modes, such as omission failures, are more frequent by a factor of 50 compared to arbitrary failures.

Also, failure persistence is an important factor in the differentiation of failure rates. While transient failures disappear without an explicit repair action, permanent failures prevail until removed by a maintenance engineer (e.g., software update in case of a software fault, replacement or repair of hardware in case of a hardware fault).

MAXIMUM NUMBER OF FAILURES

This parameter of the fault hypothesis denotes the maximum number of FCR failures, which must be handled by the system. The maximum number of failures depends on the failure rate and the recovery interval of FCRs. A prevalent assumption in many present-day safety-critical systems is the occurrence of a single failure only (e.g., TTA [48], FlexRay [62]). A failure hypothesis with this hypothesis is also frequently denoted as a "single fault hypothesis" (from a system-level point of view).

RECOVERY INTERVAL OF AN FCR

The FCR recovery interval is the maximum interval of time after a FCR failure until the FCR provides the correct service again. In case of a permanent failure, this recovery interval equals the mission time or the duration between maintenance actions. In case of a transient failure, this interval equals the sum of the failure detection latency, the FCR restart duration, and the state restoration duration [4].

5. FAULT HYPOTHESIS OF THE DECOS INTEGRATED ARCHITECTURE

This section describes the fault hypothesis of the integrated system architecture. The fault hypothesis of the complete system consists of a sub-hypothesis for hardware faults and a sub-hypothesis for software faults. We assume a single failure of a node computer or communication channel. Node computers may exhibit an arbitrary failure mode. For the communication channels, we assume that the network will not spontaneously create correct frames and will not introduce arbitrary delays when forwarding frames. For software faults, we assume zero failures of safety-critical software (including safety-critical jobs and system software) and an arbitrary number failures of non safety-critical jobs. In case of software diversity, the zero failure assumption can be relaxed to a single failure assumption. The failure modes encompass an arbitrary behavior at the execution environment and communication system.

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., wearout [63]) and external interference through physical phenomena (e.g., lightning stroke). Early and premature wearout failures are caused by the displacement of the mean and variability due to manufacturing, assembly, handling, and misapplication [64].

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 [65]. According to Constantinescu [66] the primary cause for the significant increase of the Soft Error Rate (SER) 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 [67]. This SER in chip-level design [68] is estimated to significantly increase without additional error protection mechanisms or the use of more robust technology such as Silicon on Insulator (SOI), as more and more devices are added to a processor [69]. In literature a discrimination between masked errors, correctable errors, Detected Uncorrectable Errors (DUE), and Silent Data Corruption (SDC) is made [70, 69]. For example, in [71] a SER of 114 FIT (1000 year) for SDC, 4,566

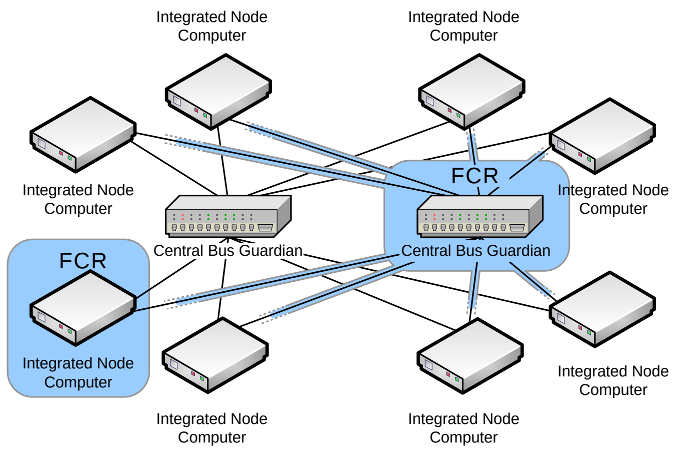

Figure 5: Hardware Fault Containment Regions

FIT (25 year MTTF) for system-kill DUE, and 11,415 FIT (10 year MTTF) for process-kill DUE is targeted for IBM servers. Xilinx, vendor of FPGA devices, supports these numbers by providing experimental data regarding the reliability of their products with respect to Single Event Upset (SEU) [72, 73]. The mean time between logic failures for the XC2V6000 device is 170 years. The MTBF for a logic error caused by SEU in an XC2V1000 device is 1000 years, or 114 FIT [74].

Fault-Containment Regions. For hardware faults, we regard each node computer (i.e., a hardware/software unit) as a FCR. Since a node computer 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 node computer. Furthermore, we assume that hardware diversity is applied to prevent common mode failures [75] 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 [76, 77]. For example, the primary flight control system of the Boeing 777 employs three dissimilar microprocessors (AMD 29050, Intel 80486, Motorola 68040).

In addition, we regard each communication channel (e.g., central bus guardian plus wiring as described in [49]) as a FCR. In case of replicated communication channels, each communication channel forms a dedicated FCR. See Figure 5 for illustration.

Failure Mode Assumption. The failure mode of a node computer is assumed to be arbitrary. Fault injection campaigns have shown that more restrictive assumptions on the failure modes of node computers such as fail silence behavior cannot be justified in the context of ultra-dependable systems [47].

For the communication channels, we use the failure mode assumption introduced in [49] stating that the network will not spontaneously create correct frames and will not introduce arbitrary delays when forwarding frames.

Failure Rates Assumptions. In the definition of the failure rates of hardware FCRs it is important to distinguish between permanent and transient failures:

- **Permanent Hardware Failure Rate.** The permanent failure rate of a FCR with respect to hardware faults is considered to be in the order of 100 FIT, i.e., about 1000 years [78].
- **Transient Hardware Failure Rate.** Motivated by literature on SER we assume that the transient failure rate of a FCR with respect to hardware faults is in the order of 10.000-100.000 FIT.

Maximum Number of Failures. In the fault hypothesis a single failure of a hardware FCR is assumed. The "single fault hypothesis" is widely accepted, both in academia and industry (e.g., TTA [48], FlexRay [62]). In case of a transient FCR failure and subsequent recovery with state restoration either a further transient or permanent FCR failure can be tolerated. By contrast, in case of a permanent FCR failure only a maintenance action can restore the initial system state.

Recovery Interval of an FCR. The duration of a transient hardware FCR failure can be assumed to be in the order of tens of milliseconds. For example in [79], the transient outage-time of an automotive steering system is required to be less than 50 ms. The same duration is also specified as the maximum outage in [80]. Correlated FCR failures, i.e., a fault affecting more than one FCR at the same time, are assumed to be experienced within a bounded interval of time. According to the ISO 7637 standard [81] the duration of an EMI burst is in the order of 10 ms.

Therefore, in case of a transient FCR failure the recovery interval is assumed to be not longer than 50 ms. For a permanent FCR failure the recovery interval equals the time it takes to bring the system into a safe state and perform the corresponding maintenance action at the service station.

SOFTWARE FAULT MODEL

This section describes the fault hypothesis for software faults. The sub-hypothesis for software faults is important for the integration of multiple jobs on an integrated node computer in the context of mixed criticality systems and independent development of application subsystems.

Fault-Containment Regions. In compliance with the model of an integrated node computer in Section , we

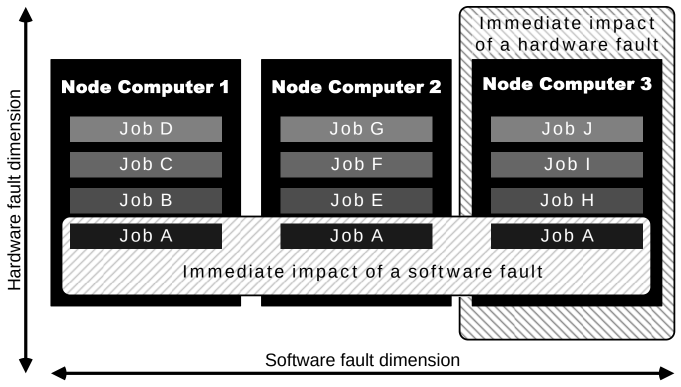

Figure 6: Immediate Impact of a Developmental Software or Interaction Fault (Horizontally Expanding Section-Lined Box) and Immediate Impact of a Hardware Fault (Vertically Expanding Section-Lined Box)

distinguish between *system software* (i.e., partition management operating system, middleware, I/O drivers) and *application software* (i.e., one or more jobs).

Since all jobs hosted on a node computer depend on the correct behavior of the system software, the jobs cannot be assumed to be unaffected by a fault affecting the system software. Therefore, *all node computers on which a particular system software is deployed represent a common FCR for software faults affecting the system software.* The system software is thus a critical resource in the integrated system. It is necessary to ensure the absence of software faults in the system software. In particular, the system software needs to be designed for validation [82] and kept simple in order to permit a thorough validation (e.g., including formal verification). Moving functionality from the system software into the partitions is a viable strategy to achieve this goal, which is similar to the well known concept of micro-kernels in operating system design [83].

For software faults affecting the application software, we regard a job as a FCR. If a job is replicated along multiple node computers as part of a fault-tolerance concept, the FCR includes all 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 a FCR consisting of replicated instances of a job is depicted in Figure 6.

The role of jobs as software FCRs holds also in case of software diversity (e.g., pursued in Airbus aircraft [77]). When software diversity is applied for addressing common mode failures, replicas are necessarily different and ideally employ different specifications in addition to separate implementations. 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 versions. Practical analyses [84, 85] of software diversity have demonstrated that diverse implementations exhibit correlation w.r.t. design faults.

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 6, the expansion of FCRs for hardware and software faults proceeds along two dimensions. In case of replicated jobs, software faults hit multiple node computers and affect well-delimited subsystems of these node computers, namely the partitions housing the replicated instances of the job. The FCR for hardware faults, on the other hand, expand within a node computer and we assume that hardware faults are delimited by a node computer. 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).

Failure Mode Assumption. A job can exhibit different failure modes for different types of used resources. Failures at the communication system comprise transmissions of incorrect messages. At the execution environment, the failure modes of a job relate to the unspecified use of computational resources. Failure modes for the communication system refer to the compliance with the interface specification of the job, while the failure modes concerning the execution environment are a statement w.r.t. component specification. In the fault hypothesis, we assume the following failure modes for the communication system and the execution environment.

- **Communication system – arbitrary value message failure.** A value message failure occurs in case the contents of a transmitted message do not comply with the interface specification. We assume arbitrary value message failures. Examples for specific value message failures of the fault hypothesis are crash/omission failures, babbling idiot, and SoS failures in the temporal domain as introduced in Section .
- **Communication system – arbitrary timing message failure.** An unspecified send instant of a message is denoted as a timing message failure. We assume arbitrary timing message failures. Masquerading failures, two-faced message contents, and SoS failures in the value domain are examples for specific timing message failures.
- **Execution environment – arbitrary timing failure.** A timing failure of a job being executed on a node computer is a violation of the temporal specification of the job, e.g., the non compliance to specified worst-case execution. In the scope of the execution environment, we assume arbitrary timing failures of jobs.
- **Execution environment – arbitrary value failure.** A value failure of a job being executed on a node computer is a violation of the value specification of the job. Possible value failures include attempted access

Safety Integrity Level	Probability of Failure per Hour
4	$\geq 10^{-9}$ to $< 10^{-8}$
3	$\geq 10^{-8}$ to $< 10^{-7}$
2	$\geq 10^{-7}$ to $< 10^{-6}$
1	$\geq 10^{-6}$ to $< 10^{-5}$

Table 1: Safety Integrity Levels According to [86]

operations to memory outside a partition or access to sensors/actuators belonging to other jobs. In the scope of the execution environment, we assume arbitrary value failures of jobs.

Failure Rates Assumptions. The presented fault hypothesis aims at integrated architectures that support the Safety Integrity Levels (SIL) 1 to 4 [86, 87]. Consequently, the failure rate assumptions presented in Table 1 apply to the fault hypothesis.

Maximum Number of Failures. For safety-critical jobs, which are part of application subsystems with reliability requirements of 10^{-7} failures/hour or better (i.e., SIL 3/SIL 4 according to IEC 61508 [88]), we assume the absence of software faults. For this purpose correct-by-construction methods have gained more and more momentum in recent years for the development of safety-critical software [89, 90]. In case of design diversity, this assumption is relaxed to the assumption of a single failure of a safety-critical job induced by a software fault.

For non safety-critical jobs with reliability requirements below 10^{-7} failures/hour (e.g., SIL 1/SIL 2 according to IEC 61508 [88]), we assume that all jobs may fail. Even in this case, the error containment mechanisms at the level of the communication and computational resources need to ensure that no error propagation towards the safety-critical jobs occurs.

Recovery Interval of an FCR. Although a software fault is by definition a permanent fault, we assume that (permanent) software faults that have evaded validation result with high probability in transient failures of software FCRs. For this reason, we assume that the system software detects job failures (e.g., violation of worst-case execution time) and performs a restart of the partition hosting the job.

A design fault in a job will manifest itself conditionally as a failure of the job depending on the execution context (e.g., inputs). As long as this context prevails, a restart of the job repeatedly causes a failure. The recovery interval denotes the duration of the time interval between the disappearance of the context for the software fault and the time at which the correct service of the job is provided again.

After the disappearance of the context, the software FCRs is expected to recover within the same time interval (50 ms) as in case of a transient failure of a hardware FCR. If the context prevails, the job does not recover.

6. CONCLUSION

The fault hypothesis of a safety-critical distributed embedded real-time system is the basis for the design and validation of the fault-tolerance algorithms, the determination of the assumption coverage, and the construction of the safety argument. In integrated systems, the fault hypothesis needs to take into account the presence of multiple application subsystems that can possess different criticalities, thus having different requirements concerning reliability. In general, such application subsystems will exhibit significant differences concerning the residue of design faults after deployment due to different development processes driven by economic constraints. In safety-critical application subsystems, the absence of design faults can no longer be shown by testing alone. The achievement of the reliability includes a rigorous development process, formal verification, and involvement of a certification agency. For non safety-critical application subsystems, certainty about the complete absence of design faults is usually economically infeasible. The level of rigidity in the development process of safety-critical applications would be too expensive. For this reason, an integrated architecture needs to prevent error propagation between application subsystems in order to prevent a design fault in a non safety-critical application subsystems from affecting safety-critical ones. The assumptions (e.g., identification of fault containment regions w.r.t. software faults) introduced in the fault hypothesis of this paper are a prerequisite for the development of these error containment mechanisms.

Apart from handling mixed criticality systems, error containment mechanisms building on top of the presented fault hypothesis are also a major step towards a seamless system integration with clear integration responsibilities. Since each of the introduced fault containment regions w.r.t. software faults can be attributed to a liable organizational entity (e.g., a specific supplier), effective error containment between fault containment regions avoids correlated system-wide failures that would blur the origin of a fault.

ACKNOWLEDGMENTS

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

REFERENCES

[1] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.

[2] R.W. Butler, J.L. Caldwell, and B.L. Di Vito. Design strategy for a formally verified reliable computing platform. In *Proc. of the 6th Annual Conference on Systems Integrity, Software Safety and Process Security*, pages 125–133, June 1991.

[3] 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.

[4] H. Kopetz. The fault hypothesis for the time-triggered architecture. In *Proc. of the IFIP World Computer Congress*, 2004.

[5] D. Powell. Failure mode assumptions and assumption coverage. In *Proc. of the 22nd IEEE Annual Int. Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 386–395, Boston, USA, July 1992.

[6] R. Obermaisser, P. Peti, B. Huber, and C. El Salloum. DECOS: An integrated time-triggered architecture. *e&i journal (journal of the Austrian professional institution for electrical and information engineering)*, 3:83–95, March 2006. Available at http://www.springerlink.com.

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

[8] AUTOSAR GbR. *AUTOSAR – Technical Overview V2.0.1*, June 2006.

[9] B. Bouyssounouse and J. Sifakis, editors. *Embedded Systems Design*. Springer Verlag, 2005.

[10] P. Giusto, A. Ferrari, L. Lavagno, J.-Y. Brunel, E. Fourgeau, and A. Sangiovanni-Vincentelli. Automotive virtual integration platforms: why's, what's, and how's. In *Proc. of the IEEE Int. Conference on Computer Design: VLSI in Computers and Processors*, pages 370–378, September 2002.

[11] J. Rushby. Modular certification. Technical report, Computer Science Laboratory SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, USA, September 2001.

[12] H.A. Simon. *The Sciences of the Artificial*. MIT Press, 1996.

[13] J. Sifakis. A framework for component-based construction. In *Proc. of 3rd IEEE Int. Conference on Software Engineering and Formal Methods (SEFM05)*, pages 293–300, September 2005.

[14] H. Kopetz and R. Obermaisser. Temporal composability. *Computing & Control Engineering Journal*, 13:156–162, August 2002.

[15] M. Short and M.J. Pont. Hardware in the loop simulation of embedded automotive control system. In *Proc. of the 8th Int. IEEE Conference on Intelligent Transportation Systems*, pages 426–431, Vienna, Austria, September 2005.

[16] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, and T. Scharnhorst. AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures. In *Proceedings of the Convergence Int. Congress & Exposition On Transportation Electronics*, Detroit, MI, USA, October 2004. SAE. 2004-21-0042.

[17] Th. Scharnhorst, H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, P. Heitkämper, J. Leflour, J.-L. Mate, and K. Nishikawa. AUTOSAR - challenges and achievements 2005. In *VDI Berichte 1907*. Verein Deutscher Ingenieure, 2005.

[18] 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.

[19] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[20] A. W. Brown and K. C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, September/ October 1998.

[21] H.D. Heitzer. Development of a fault-tolerant steer-by-wire steering system. *Auto Technology*, 4:56–60, April 2003.

[22] P. Peti, R. Obermaisser, F. Tagliabo, A. Marino, and S. Cerchio. An integrated architecture for future car generations. In *Proc. of the 8th IEEE Int. Symposium on Object-oriented Real-time distributed Computing*, May 2005.

[23] 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.

[24] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, 4th edition, September 2000.

[25] B. Huber, P. Peti, R. Obermaisser, and C. El Salloum. Using RTAI/LXRT for partitioning in a prototype implementation of the DECOS architecture. In *Proc. of the Third Int. Workshop on Intelligent Solutions in Embedded Systems*, May 2005.

[26] H. Kopetz. *Specification of the TTP/C Protocol*. TT-Tech, Schönbrunner Straße 7, A-1040 Vienna, July 1999. Available at http://www.ttpforum.org.

[27] FlexRay Consortium. BMW AG, DaimlerChrysler AG, General Motors Corporation, Freescale GmbH, Philips GmbH, Robert Bosch GmbH, and Volkswagen AG. *FlexRay Communications System Protocol Specification Version 2.1*, May 2005.

[28] K. Hoyme and K. Driscoll. SAFEbus. *IEEE Aerospace and Electronic Systems Magazine*, 8:34–39, March 1993.

[29] J. Rushby. Bus architectures for safety-critical embedded systems. In Tom Henzinger and Christoph Kirsch, editors, *Proc. of the First Workshop on Embedded Software (EMSOFT 2001)*, volume 2211 of *Lecture Notes in Computer Science*, pages 306–323, Lake Tahoe, CA, October 2001. Springer-Verlag.

[30] L. Kinnan, J. Wlad, and P. Rogers. Porting applications to an ARINC 653 compliant IMA platform using VxWorks as an example. In *Proc. of the 23rd Digital Avionics Systems Conference*, volume 2, pages 10.B.1–10.1–8, October 2004.

[31] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Specification 653-1 (Draft 3): Avionics Application Software Standard Interface*, July 2003.

[32] A. Subbarao. The technology behind LynxOS v4.0's Linux ABI compatibility. Technical report, LinuxDevices.com, June 2002.

[33] R. Obermaisser, P. Peti, and H. Kopetz. Virtual networks in an integrated time-triggered architecture. In *Proc. of the 10th IEEE Int. Workshop on Object-oriented Real-time Dependable Systems (WORDS2005)*, pages 241–253, Sedona, Arizona, February 2005.

[34] LIN Consortium. *LIN Specification Package Revision 2.0*, September 2003.

[35] T. Rolina. Past, present, and future of real-time embedded automotive software: A close look at basic concepts of AUTOSAR. In *Proc. of SAE World Congress*, Detroit, Michigan, April 2006.

[36] B. Witwer. Systems integration of the 777 airplane information management system (aims). *IEEE Aerospace and Electronic Systems Magazine*, 11(4):17–21, April 1996.

[37] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Specification 653: Avionics Application Software Standard Interface, Part 1 - Required Services*, March 2006.

[38] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Specification 429: Digital Information Transfer System*, November 2001.

[39] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Specification 664 (Draft): Aircraft Data Network Part 7 – Deterministic Networks*, May 2003.

[40] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Specification 629: Multi-Transmitter Data Bus – Part 1: Technical Description*, November 1991.

[41] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Specification 659: Backplane Data Bus*, December 1993.

[42] P. Peti and R. Obermaisser. A diagnostic framework for integrated time-triggered architectures. In *Proc. of the 9th IEEE Int. Symposium on Object-oriented Real-time distributed Computing*, April 2006.

[43] R. Obermaisser and P. Peti. Specification and execution of gateways in integrated architectures. In *Proc. of the 10th IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA)*, Catania, Italy, September 2005. IEEE.

[44] TTTech Computertechnik AG, Schönbrunner Strasse 7, A-1040 Vienna, Austria. *Time-Triggered Protocol TTP/C – High Level Specification Document*, July 2002.

[45] FlexRay Consortium. *FlexRay Communications System Protocol Specification Version 2.1*, May 2005.

[46] J.H. Lala and R.E. Harper. Architectural principles for safety-critical real-time applications. *Proc. of the IEEE*, 82:25–40, January 1994.

[47] A. Ademaj. *Assessment of Error Detection Mechanisms of the Time-Triggered Architecture Using Fault Injection*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2003.

[48] H. Kopetz and G. Bauer. The time-triggered architecture. *IEEE Special Issue on Modeling and Design of Embedded Software*, January 2003.

[49] G. Bauer, H. Kopetz, and W. Steiner. The central guardian approach to enforce fault isolation in a time-triggered system. In *Proc. of the 6th Int. Symposium on Autonomous Decentralized Systems (ISADS 2003)*, pages 37–44, Pisa, Italy, April 2003.

[50] A. Avizienis, J.C. Laprie, and B. Randell. Fundamental concepts of dependability. Research Report 01-145, LAAS-CNRS, Toulouse, France, April 2001.

[51] B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys*, 10(2):123–165, 1978.

[52] A. Avizienis. Fault-tolerance and fault-intolerance: Complementary approaches to reliable computing. In *Proc. of the Int. conference on Reliable software*, pages 458–464, 1975.

[53] P.A. Lee and T. Anderson. *Fault Tolerance Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer Verlag, 1990.

[54] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[55] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[56] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[57] H. Kopetz. Fault containment and error detection in the time-triggered architecture. In *Proc. of the Sixth Int. Symposium on Autonomous Decentralized Systems*, April 2003.

[58] Robert Bosch Gmbh, Stuttgart, Germany. *CAN Specification, Version 2.0*, 1991.

[59] J. Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, September 2001.

[60] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Int. Computer Science Series. Addison-Wesley, Reading, MA, USA, second edition, 1994.

[61] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, and G. Leber. Integration and comparison of three physical fault injection techniques. In B. Randell, J. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictably Dependable Computing Systems*, pages 309–327. Springer Verlag, heidelberg edition, 1995.

[62] FlexRay Consortium. BMW AG, DaimlerChrysler AG, General Motors Corporation, Freescale GmbH, Philips GmbH, Robert Bosch GmbH, and Volkswagen AG. *FlexRay Requirements Specification Version 2.1*, December 2005.

[63] A. Ramakrishnan. *The Avionics Handbook*, chapter Electronic Hardware Reliability. CRC Press LCC, 2001.

[64] M. Pecht. Electronic reliability engineering in the 21st century. In *Proc. of 2001 Int. Symposium on Electronic Materials and Packaging*, pages 1–7. IEEE, 2001.

[65] 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.

[66] C. Constantinescu. Impact of deep submicron technology on dependability of VLSI circuits. In *Proc. of the Int. Conference on Dependable Systems and Networks*, pages 205–209. IEEE, 2002.

[67] L. Lev and P. Chao. Down to the wire. Technical report, Cadence Design Systems, Inc., San Jose, CA, USA, 2002.

[68] R. Baumann. Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22(3):258–266, May 2005.

[69] S.S. Mukherjee, J. Emer, and S.K. Reinhardt. The soft error problem: an architectural perspective. In *Proc. of the 11th Int. Symposium on High-Performance Computer Architecture*, pages 243–247, February 2005.

[70] H.T. Nguyen, Y. Yagil, N. Seifert, and M. Reitsma. Chip-level soft error estimation method. *IEEE Transactions on Device and Materials Reliability*, 5(3):365 – 381, September 2005.

[71] D.C. Bossen. CMOS soft errors and server design. *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, April 2002.

[72] E. Normand. Single-event effects in avionics. *IEEE Transactions on Nuclear Science*, 43(2):461–474, April 1996.

[73] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43(6):2742–2750, December 1996.

[74] P. Alfke and A. Lesea. A thousand years between single-event upset failures. Technical report, XILINX, 2003.

[75] L. Kaufman, S. Bhide, and B. Johnson. Modeling of common-mode failures in digital embedded systems. In *Proc. of the Reliability and Maintainability Symposium*, pages 350–357, Los Angeles, CA, USA, 2000. IEEE Press.

[76] Y.C. Yeh. Design considerations in Boeing 777 fly-by-wire computers. In *Proc. of the 3rd IEEE Int. High-Assurance Systems Engineering Symposium*, pages 64–72, November 1998.

[77] D. Briere and P. Traverse. AIRBUS A320/A330/A340 electrical flight controls - a family of fault-tolerant systems. In *Proc. of the Twenty-Third Int. Symposium on Fault-Tolerant Computing*, pages 616–623, June 1993.

[78] 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.

[79] G. Heiner and T. Thurner. Time-triggered architecture for safety-related distributed real-time systems in transportation systems. In *Proc. of the Twenty-Eighth Annual Int. Symposium on Fault-Tolerant Computing*, pages 402–407, June 1998.

[80] US Department of Transportation Federal Aviation Administration. System design and analysis, ac no. 25.1309-1a. Technical report, 1988.

[81] Int. Standardization Organisation, ISO 7637. *Road vehicles – Electrical disturbances from conduction and coupling*, 1995.

[82] S.C. Johnson and R.W. Butler. Design for validation. *IEEE Aerospace and Electronic Systems Magazine*, 7(1):38–43, January 1992.

[83] J. Liedtke. On micro-kernel construction. In *SOSP '95: Proc. of the fifteenth ACM symposium on Operating systems principles*, pages 237–250, New York, NY, USA, 1995. ACM Press.

[84] 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 Int. Symposiun on Fault Tolerant Computing*, pages 15–22, 1988.

[85] B. Littlewood, P. Popov, and L. Strigini. Modeling software design diversity: a review. *ACM Comput. Surv.*, 33(2):177–208, 2001.

[86] IEC: Int. Electrotechnical Commission. *IEC 61508-1: General Requirments*, 1998.

[87] D.J. Smith and K.G. Simpson. *Funcational Safety*. Elsevier, second edition, 2004.

[88] IEC: Int. Electrotechnical Commission. *IEC 61508-7: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems – Part 7: Overview of Techniques and Measures*, 1999.

[89] G. Berry. The foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000.

[90] B. Dion. Correct-by-construction methods for the development of safety-critical applications. In *SAE 2004 World Congress & Exhibition*, Detroit, MI, USA, March 2004. SAE.