Chapter 11
System Design

Overview This chapter on architecture design starts with a discussion on design in general. The designer must get a deep insight into all different aspects of the problem domain before she/he can design a proper structure for the application. In computer system design, the most important goal is controlling the complexity of the evolving artifact. A thorough analysis of the requirements and constraints limits the design space and avoids the investigation of unrealistic design alternatives. Any kind of structure restricts the design space and has a negative impact on the performance of a system, which must be carefully evaluated in real-time systems. The central step in the development of an architecture is concerned with the allocation of functions to nearly decomposable *clusters of components*. Components should have a high internal cohesion and simple external interfaces. In the following, different design styles such as *model-based design* and *component-based design* are discussed. The design of safety-critical systems starts with the *safety analysis* such as fault tree analysis and/or failure mode and effect analysis (FMEA) of the envisioned application, and the development of a convincing *safety case*. Different standards that must be observed in the design of safety-critical system are described, such as the IEC 61508 for electric and electronic equipment and the ARINC DO 178B standard for airborne equipment software. The elimination of all design errors, e.g., software errors or hardware errata of a large safety-critical system is a major challenge. Design diversity can help to mitigate the problem of design errors. The final section of this chapter is devoted to the topic of *design for maintainability* in order to reduce the overall life-cycle cost. Maintainability of software is needed to correct design errors in the software and to adapt the software to the never-ending needs of an evolving application scenario. If an embedded system is connected to the Internet, a new threat, the remote attack of the system by an intruder to exploit existing vulnerabilities, must be considered. The secure download of a new software version via the Internet is an essential functionality that should be supported by any embedded system connected to the Internet.

H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Real-Time Systems Series, DOI 10.1007/978-1-4419-8237-7_11,

11.1 System Design

11.1.1 The Design Process

Design is an inherently creative activity, where both the *intuitive* and the *rational* problem solving systems of the human mind are heavily involved. There is a common core to design activities in many diverse fields: building design, product design, and computer system design are all closely related. The designer must find a solution that *accommodates a variety of seemingly conflicting goals* to solve an often ill-specified design problem. In the end, what differentiates a good design from a bad design is often liable to subjective judgment.

> **Example:** Consider the design of an automobile. An automobile is a complex mass-production product that is composed of a number of sophisticated subsystems (e.g., engine, transmission, chassis, etc.). Each of these subsystems itself contains hundreds of different components that must meet given constraints: functionality, efficiency, geometrical form, weight, dependability, and minimal cost. All these components must cooperate and interact smoothly, to provide the *emergent* transportation service and the *look and feel* that the customer expects from the system *car*.

During the *purpose analysis phase*, the organizational goals and the economic and technical constraints of an envisioned computer solution are established. If the evaluation at the end of this phase results in a *go ahead* decision, a project team is formed to start the requirement analysis and the architecture design phase. There are two opposing empirical views how to proceed in these first life cycle phases when designing a large system:

1. A disciplined sequential approach, where every life-cycle phase is thoroughly completed and validated before the next one is started (*grand design*)
2. A rapid-prototyping approach, where the implementation of a key part of the solution is started before the requirements analysis has been completed (*rapid prototyping*)

The rationale for the *grand design* is that a detailed and unbiased specification of the complete problem (the *What*?) must be available before a particular solution (the *How?*) is designed. The difficulty with *grand design* is that there are no clear *stopping rules*. The analysis and understanding of a large problem is never complete and there are always good arguments for asking more questions concerning the requirements before starting with the *real* design work. Furthermore, the world evolves while the analysis is done, changing the original scenario. The phrase *paralysis by analysis* has been coined to point to this danger.

The rationale for the *rapid prototyping* approach assumes that, by investigating a particular solution at an early stage, a lot is learned about the problem space. The difficulties met during the search for a concrete solution guide the designer in asking the right questions about the requirements. The dilemma of rapid prototyping is that ad hoc implementations are developed with great expense. Since the first

prototype does address limited aspects of the design problem only, it is often necessary to completely discard the first prototypes and to start all over again.

Both sides have valid arguments that suggest the following compromise: In the architecture design phase, a key designer should try to get a good understanding of the architectural properties, leaving detailed issues that affect only the internals of a subsystem open. If it is not clear how to solve a particular problem, then a preliminary prototype of the most difficult part should be investigated with the explicit intent of discarding the solution if the looked-for insight has been gained. In his recent book [Bro10], Fred Brook states that *conceptual integrity of a design is the result of a single mind.*

Some years ago, Peters [Pet79] argued in a paper about design that design belongs to the set of *wicked problems*. Wicked problems are described by the following characteristics:

1. A wicked problem cannot be stated in a definite way, abstracted from its environment. Whenever one tries to isolate a wicked problem from its surroundings, the problem loses its peculiarity. Every wicked problem is somehow unique and cannot be treated in the abstract.
2. A wicked problems cannot be specified without having a solution in mind. The distinction between specification (*what*?) and implementation (*how*?) is not as easy as is often proclaimed in academia.
3. Solutions to wicked problems have no stopping rule: for any given solution, there is always a better solution. There are always good arguments to learn more about the requirements to produce a better design.
4. Solutions to wicked problems cannot be right or wrong; they can only be *better* or *worse*.
5. There is no definite test for the solution to a wicked problem: whenever a test is *successfully* passed, it is still possible that the solution will fail in some other way.

11.1.2 The Role of Constraints

Every design is embedded in a design space that is bounded by a set of known and unknown *constraints*. In some sense, constraints are *antonyms* to requirements. It is good practice to start a design by capturing the constraints and classifying them into *soft constraints*, *hard constraints*, and *limiting constraints*. A *soft constraint* is a desired but not obligatory constraint. A *hard constraint* is a given mandatory constraint that must not be neglected. A *limiting constraint* is a constraint that limits the utility of a design.

Example: In building a house, the mandatory construction code of the area is a *hard constraint*, the orientation of the rooms and windows is a *soft constraint*, while the construction cost may be a *limiting constraint*.

Constraints limit the design space and help the designer to avoid the exploration of design alternatives that are unrealistic in the given environment. Constraints are thus our friends, not our adversaries. Special attention must be paid to the limiting constraints, since these constraints are instrumental for determining the value of a design for the client. It is good practice to precisely monitor the limiting constraints as a design proceeds.

Example: In the European research initiative ARTEMIS that intends to develop a cross-domain architecture for embedded systems, the first step was the capture and documentation of the requirements and constraints that such an architecture must satisfy. These constraints are published in [Art06].

11.1.3 System Design Versus Software Design

In the early days of *computer-application design*, the focus of design was on the *functional aspects* of software, with little regard for the *nonfunctional* properties of the computations that are generated by the software, such as *timing*, *energy efficiency*, or *fault tolerance*. This focus has led to *software design methods* – still prevalent today – that concentrate on the data transformation aspects of a program with little regard for the temporal or energy dimension.

Example: A critical constraint in the design of a smart phone is the expected life of a battery load. This non-functional constraint is overlooked if the focus during the design is only on the functional properties of the design.

Software per se is a *plan* describing the operations of a real or virtual machine. A plan by itself (without a machine) does not have *any temporal dimension*, cannot have *state* (which depends on a precise notion of real-time – see Sect. 4.2.1) and has no *behavior*. Only the *combination* of software and the targeted machine, the *platform*, produces behavior. This is one of the reasons why we consider the *component* and not the *job* (see Sect. 4.2.2) as the primitive construct at the level of architecture design of an embedded system.

The complete functional and temporal specification of the *behavior of a job* (i.e., the software for a machine) is much more complicated than the specification of the *behavior of a component*. In addition to the four message interfaces of a component described in Sect. 4.4, the complete specification of a job must include the functional and temporal properties of the *API* (application programming interface) of the job to the targeted virtual or real machine [Szy99]. If the underlying machine is *virtual*, e.g., an execution environment that is built *bottom-up* by some other software, e.g., a *hypervisor*, the temporal properties of this virtual machine depend on the software design of the hypervisor and the hardware performance of the physical machine. But even without a hypervisor, the temporal hardware performance of many of today's sophisticated sequential processors with multiple levels of caching and speculative execution is difficult to specify. Considering the implications of *Pollack's rule* (see Sect. 8.3.2), we conjecture that in the domain of embedded real-time systems *predictable sequential processors* combined with the

appropriate system and application software will form the IP-cores, the *components*, of the envisioned multiprocessor systems-on-chips (MPSoC) of the embedded system of the future.

The intended behavior of a component can be realized by different implementation technologies:

1. By designing software for a programmable computer, resulting in a flexible component consisting of a local operating system with middleware and application software modules.
2. By developing software for a field-programmable gate array (FPGA) that implements the component's functionality by the proper interconnection of a set of highly concurrent logic elements.
3. By developing an application specific integrated circuit (ASIC) that implements the functionality of the component directly in hardware.

Viewed from the outside, the services of a component must be *agnostic* of the chosen implementation technology. Only then it is possible to change the implementation of a component without any effects at the system level. However, from the point of view of some of the non-functional component characteristics such as energy consumption, silicon real-estate requirements, flexibility to change, or non-recurring development costs, different component implementations have vastly different characteristics. In a number of applications it is desired to develop at first a hardware-agnostic model of the services of a component at the architecture level and to postpone the detailed decisions about the final implementation technology of the component to a later stage.

Example: In a product for mass-market consumer appliance, it makes sense to first develop a prototype of a component in software-on-a CPU and to decide later, after the market acceptance of the product has been established, to shift the implementation to an FPGA or ASIC.

11.2 Design Phases

Design is a creative holistic human activity that cannot be reduced to following a set of rules out of a design rule-book. Design is an art, supplemented by scientific principles. It is therefore in vain to try to establish a complete set of design rules and to develop a fully automated design environment. Design tools can assist a designer in handling and representing the design information and can help in the analysis of design problems. They can, however, never replace a creative designer.

In theory, the design process should be structured into a set of distinct phases: purpose analysis, requirements capture, architecture design, detailed component design and implementation, component validation, component integration, system validation, and finally system commissioning. In practice, such a strict sequential decomposition of the design process is hardly possible, since the full scope of a new design problem is not comprehended until the design process is well under its way, requiring frequent iterations among the design phases.

The focus of this chapter is on the architecture design phases, while the validation phases are covered in Chap. 12.

11.2.1 Purpose Analysis

Every rational design is driven by a given *purpose*. The purpose puts the design into the wider context of user expectations and economic justification and thus precedes the requirements. *Purpose analysis*, i.e., the analysis why a new system is needed and what is the ultimate goal of a design must precede the requirements analysis, which already limits the scope of analysis and directs the design effort to a specific direction. Critical purpose analysis is needed in order to put the requirements into the proper perspective.

> **Example:** The purpose of acquiring a car is to provide a transportation service. There are other means of transportation, e.g., public transport, which should be considered in the purpose analysis phase.

In every project, there is an ongoing conflict between *what is desired* and *what can be done* within the given technical and economic constraints. A good understanding and documentation of these technical and economic constraints reduces the design space and helps to avoid exploring unrealistic design alternatives.

11.2.2 Requirements Capture

The focus of the requirements phase is to get a good understanding and a concise documentation of the requirements and constraints of the essential system functions that provide the economic justification of the project. There is always the temptation to get sidetracked by irrelevant details about representational issues that obscure the picture of the whole. Many people find it is easier to work on a well-specified detailed side problem than to keep focus on the critical system issues. It requires an experienced designer to decide between a *side problem* and a *critical system issue*.

Every requirement must be accompanied by an acceptance criterion that allows to measure, at the end of the project, whether the requirement has been met. If it is not possible to define a distinct acceptance test for a requirement, then the requirement cannot be very important: it can never be decided whether the implementation is meeting this requirement or not. A critical designer will always be suspicious of postulated requirements that cannot be substantiated by a rational chain of arguments that, at the end, leads to a measurable contribution of the stated requirement to the purpose of the system.

In the domain of embedded systems, a number of *representation standards* and *tools* have been developed to support the system engineer. Standards for the

uniform representation of requirements are of particular importance, since they simplify the communication among designers and users. The recent extension of the UML (Unified Modeling Language) with the MARTE (Modeling and Analysis of Real-Time Embedded Systems) profile provides a widely accepted standard for the representation of real-time requirements [OMG08].

11.2.3 Architecture Design

After the essential requirements have been captured and documented, the most crucial phase of the life cycle, the design of the system architecture, follows. *Complex systems will evolve from simple systems much more rapidly if there are stable intermediate forms than if there are not* [Sim81]. Stable intermediate forms are encapsulated by small and stable interfaces that restrict the interactions among the subsystems. In the context of distributed real-time systems, the architecture design establishes the decomposition of the overall systems into *clusters* and *components*, the linking interfaces of the *components*, and the *message communication* among the components.

In general, introducing structure restricts the design space and may have a negative impact on the performance of a system. The more rigid and stable the structure, the more notable the observed reduction in performance will be. The key issue is to find the *most appropriate* structure where the performance penalties are outweighed by the other desirable properties of the structure, such as composability, understandability, energy efficiency, maintainability, and the ease of implementing fault-tolerance or of adding new functionality in an evolving system and environment.

11.2.4 Design of Components

At the end of the architectural design phase, the requirements have been allocated to components, and the *linking interfaces* (LIFs) of the components are precisely specified in the value domain and in the temporal domain. The design effort can now be broken down into a set of concurrent design activities, each one focusing on the design, implementation, and testing of a single component.

The detailed design of the *local interface* (see Sect. 4.4.5) between a component and its local environment (e.g., the controlled object or other clusters) is not covered in the architectural design phase, since only the *semantics*, but not the *syntax* of these local interfaces are needed for the cluster-LIF specification of the components. It is up to the detailed design of a component to specify and implement these local interfaces. In some cases, such as the design of the concrete man-machine interface for the operator, this can be a major activity.

The detailed steps that have to be taken in order to implement the design of a component depend on the chosen implementation technology. If the services of

a component are implemented by a *software-on-a-CPU* design, then the necessary design steps will differ radically from a design that targets an ASIC as its final outcome. Since the focus of this book is on the topic of architecture design of embedded systems, we do not cover the detailed component implementation techniques for the different implementation technologies at any length.

11.3 Design Styles

11.3.1 Model-Based Design

In Chapter two of this book we emphasize the role of model building for the understanding of any real-world scenario. Model-based design is a design method that establishes a useful framework for the development and integration of executable models of the controlled object and of the controlling computer system early in the design cycle.

After finishing the *purpose analysis* of a control system, in *model-based design*, executable high-level models of the *controlled object* (the plant) and the *controlling computer system* are developed in order that the dynamic interaction of these models can be studied at a high level of abstraction.

The first step in model-based design is concerned with the identification and mathematical modeling of the dynamics of the controlled object, i.e., the plant. Where possible, the results of the plant model are compared with experimental data from the operation of a real plant in order to validate the *faithfulness* of the plant model. In a second step, the mathematical analysis of the dynamic plant model is used for the synthesis of control algorithms that are tuned to the dynamics of the given plant model. In a third step, the executable plant model and the executable controlling computer model are integrated in a simulated environment, such that the correct interplay between the models can be validated and the quality of the considered control algorithms can be investigated in relation to the plant model. Although this simulation will often be operated in simulated time (*SIL – software in the loop*), it is important that the phase relationship between the messages exchanged between the plant model und the controlling computer model in the simulated environment and the (later) real-time control environment is exactly the same [Per10]. This exact phase relationship among the messages ensures that the *message order* in the simulations and in the target system will be alike. Finally, in the fourth phase the controlling computer system model is translated, possibly automatically, to the target execution environment of the control computers.

In *hardware-in-the-loop simulations* (*HIL*), the simulation models must be executed in real-time, since subsystems of the simulation are formed by the final target hardware.

Model-based design makes it possible to study the system performance not only under normal operating conditions, but also in *rare-event* situations, e.g., when a

critical part of the system has failed. During the simulation it is possible to tune the control algorithms such that a safe operation of the plant in a *rare event scenario* can be maintained. Furthermore, a model-based design environment can be used for automated testing and the training of plant operators.

> **Example:** It is a standard procedure to train pilots on a simulator in order to get them acquainted with the necessary control actions in case of a rare-event incident that cannot be reproduced easily during the flight of a real airplane.

A key issue in model-based design focuses on the specification of the linking interface (LIF) between the plant model and the controlling computer system model. As already discussed in Sect. 4.4.5, this interface specification must cover the value dimension and the temporal dimension of the messages that cross the LIF. The semantic interface models must be presented in an executable form, such that the simulation of the complete control system can be performed on a high level of abstraction and the automatic generation of the code for the target control system is supported. A widely used tool environment for model-based design is the MATLAB design environment [Att09].

11.3.2 Component-Based Design

In many engineering disciplines, large systems are built from prefabricated components with known and validated properties. Components are connected via stable, understandable, and standardized interfaces. The system engineer has knowledge about the global properties of the components – as they relate to the system functions – and of the detailed specification of the component interfaces. Knowledge about the internal design and implementation of the components is neither needed, nor available in many cases. A prerequisite for such a constructive approach to system building is that the validated properties of the components are not affected by the system integration. This composability requirement is an important constraint for the selection of a platform for the component-based design of large distributed real-time systems.

Component-based design is a *meet-in-the middle* design method. On the side the *functional and temporal requirements* of the components are derived top-down from the desired application functions. On the other side, the *functional and temporal capabilities* of the available components are provided by the component specifications (bottom up). During the design process, a proper match between component requirements and component capabilities must be established. If there is no component available that meets the requirements, a new component must be developed.

A prerequisite of any component-based design is a crystal clear component concept that supports the precise specification of the services that are delivered and acquired across the component interfaces. The notion of component as a hardware-software unit, introduced in Sect. 4.1.1 provides for such a component

concept. In many non-real-time applications, a software-unit is considered to form a component. In real-time systems, where the temporal properties of components are as important as the value properties, the notion of a software component is of questionable utility, since no temporal capabilities can be assigned to software without associating the software with a concrete machine. The specification of the temporal properties of the API (application programming interface) between the application software and the execution environment (middleware, operating system) of the concrete machine is so involved that a simple specification of the temporal properties of the API is hardly possible. If the mental effort needed to understand the specification of the component interfaces is in the same order of magnitude as the effort needed to understand the internals of the component operation, the abstraction of a component does not make sense any more.

The temporal capabilities of a (hardware-software) component are determined by the frequency of the oscillator that drives the component hardware. According to Sect. 8.2.3, this frequency can be lowered if the voltage is lowered (*voltage-frequency scaling*), resulting in substantial savings of the energy required to perform the computation of the component at the expense of extending the real-time needed for the execution of the computation. A holistic resource scheduler that is aware of the temporal needs and the energy requirements can match the temporal capabilities of a component to the temporal requirements of the application, thus saving energy. Energy saving is very important in mobile battery operated devices, an important sector of the embedded systems market.

11.3.3 Architecture Design Languages

The representation of the platform independent model, the PIM (the design at the architectural level), e.g., in the form of components and messages, requires a notation that is apt for this purpose.

In 2007, the Object Management Group (OMG) extended the Unified Modeling Language (UML) by a profile called MARTE (Modeling and Analysis of Real-Time and Embedded system) that extends UML to support the specification, design and analysis of embedded real-time systems at the architectural level [OMG08]. UML-MARTE targets the modeling of both the software part and of the hardware part of an embedded system. The core concepts of UML-MARTE are expressed in two packages, the *foundation package* that is concerned with structural models and the *causality package* that focuses on behavioral modeling and timing aspects. In UML-MARTE the fundamental unit of behavior is called an *action* that transforms a set of inputs into a set of outputs, taking a specified duration of real-time. Behaviors are composed out of *actions* and are initiated by *triggers*. The UML-MARTE specification contains a special section on the modeling of time. It distinguishes between three different kinds of *time abstractions*: (1) *logical time* (called *causal*/*temporal*) that is only concerned with temporal order without any notion of a temporal metric between events, (2) *discrete time* (called *clocked-*

synchronous) where the continuum of time is partitioned by a clock into a set of ordered *granules* and where actions may be performed within a granule, and (3) *real time* (called *physical/real time*) where the progression of real-time is precisely modeled. For a detailed description of UML MARTE refer to [OMG08].

Another example of an architecture design language is the AADL (Architecture Analysis and Design Language) developed at the Carnegie Mellon University Software Engineering Institute and standardized in 2004 by the Society of Automotive Engineers (SAE). AADL has been designed to specify and analyze the architecture of large embedded real-time systems. The core concept of AADL is the notion of a *component* that interacts with other components across interfaces. An AADL component is a *software unit* enhanced by attributes that capture the characteristics of the machine that is bound to the software unit, such that timing requirements and the worst-case execution time (WCET) of a computation can be expressed. AADL components interact exclusively though defined interfaces that are bound to each other by *declared connections*. AADL supports a graphical user interface and contains language constructs that are concerned with the implementation of components and the grouping of components into more abstract units called *packages*. There are tools available to analyze an AADL design from the point of view of timing and reliability. For a detailed description of AADL refer to [Fei06].

GIOTTO [Hen03] is a language for representing the design of a time-triggered embedded system at the architectural level. GIOTTO provides for intermediate abstractions that allow the design engineer to annotate the functional programming modules with temporal attributes that are derived from the high-level stability analysis of the control loops. In the final development step, the assignment of the software modules to the target architecture, these annotations are constraints that must be considered by the GIOTTO compiler.

System C is an extension of C++ that enables the seamless hardware/software co-simulation of a design at the architectural level, and provides for a step-by-step refinement of a design down to the register transfer level of a hardware implementation or to a C program [Bla09]. System C is well suited to represent the functionality of a design at the PIM level.

11.3.4 Test of a Decomposition

We do not know how to measure the quality of the result of the architecture design phase on an absolute scale. The best we can hope to achieve is to establish a set of guidelines and checklists that facilitate the comparison of two design alternatives relative to each other. It is good practice to develop a project-specific checklist for the comparison of design alternatives at the beginning of a project. The guidelines and checklists presented in this section can serve as a starting point for such a project-specific checklist.

Functional Coherence. A component should implement a self-contained function with high internal coherence and low external interface complexity. If the

component is a gateway, i.e., it processes input/output signals from its environment, only the abstract message interface, the LIF, to the cluster and not the local interface to the environment (see Sect. 4.3.1) is of concern at the level of architecture design. The following list of questions is intended to help determine the functional coherence and the interface complexity of a component:

1. Does the component implement a self-contained function?
2. Is the g-state of the component well defined?
3. Is it sufficient to provide a single level of error recovery after any failure, i.e., a restart of the whole component? A need for a multi-level error recovery is always an indication of a weak functional coherence.
4. Are there any control signals crossing the message interface or is the interface of a component to its environment a strict data-sharing interface? A strict data-sharing interface is simpler and should therefore be preferred. Whenever possible, try to keep the temporal control within the subsystem that you are designing (e.g., on input, *information pull* is preferable over *information push* see Sect. 4.4.1)!
5. How many different data elements are passed across the message interface? Are these data elements part of the interface model of the component? What are the timing requirements?
6. Are there any phase-sensitive data elements passed across the message interface?

Testability. Since a component implements a single function, it must be possible to test the component in isolation. The following questions should help to evaluate the testability of a component:

1. Are the temporal as well as the value properties of the message interface precisely specified such that they can be simulated in a test environment?
2. Is it possible to observe all input/output messages and the g-state of a component without the probe effect?
3. Is it possible to set the g-state of a component from the outside to reduce the number of test sequences?
4. Is the component software deterministic, such that the same input cases will always lead to the same results?
5. What are the procedures to test the fault-tolerance mechanisms of the component?
6. Is it possible to implement an effective *built-in self test* into the component?

Dependability: The following checklist of questions refers to the dependability aspects of a design:

1. What is the effect of the worst malicious failure of the component to the rest of the cluster? How is it detected? How does this failure affect the minimum performance criterion?
2. How is the rest of the cluster protected from a faulty component?
3. In case the communication system fails completely, what is the local control strategy of a component to maintain a safe state?

4. How long does it take other components of the cluster to detect a component failure? A short error-detection latency simplifies the error handling drastically.
5. How long does it take to restart a component after a failure? Focus on the fast recovery from any kind of a single fault – A single Byzantine fault [Dri03]. The zero fault case takes care of itself and the two or more independent Byzantine fault case is expensive, unlikely to occur, and unlikely to succeed. How complex is the recovery?
6. Are the normal operating functions and the safety functions implemented in different components, such that they are in different FCUs?
7. How stable is the message interface with respect to anticipated change requirements? What is the probability and impact of changes of a component on the rest of the cluster?

Energy and Power. Energy consumption is a critical non-functional parameter of a mobile device. Power control helps to reduce the silicon die temperature and consequently the failure rate of devices:

1. What is the energy budget of each component?
2. What is the peak power dissipation? How will peak-power effect the temperature and the reliability of the device.
3. Do different components of an FCU have different power sources to reduce the possibility of common mode failures induced by the power supply? Is there a possibility of a common mode failure via the grounding system (e.g., lightning stroke)? Are the FCUs of an FTU electrically isolated?

Physical Characteristics. There are many possibilities to introduce common-mode failures by a careless physical installation. The following list of questions should help to check for these:

1. Are mechanical interfaces of the replaceable units specified, and do these mechanical boundaries of replaceable units coincide with the diagnostic boundaries?
2. Are the FCUs of an FTU (see Sect. 6.4.2) mounted at different physical locations, such that spatial proximity faults (e.g., a common mode external fault such as water, EMI, and mechanical damage in case of an accident) will not destroy more than one FCU?
3. What are the cabling requirements? What are the consequences of transient faults caused by EMI interference via the cabling or by bad contacts?
4. What are the environmental conditions (temperature, shock, and dust) of the component? Are they in agreement with the component specifications?

11.4 Design of Safety-Critical Systems

The economic and technological success of embedded systems in many applications leads to an increased deployment of computer systems in domains where a computer failure can have severe consequences. A computer system becomes *safety-critical*

(or *hard real-time*) when a failure of the computer system can have catastrophic consequences, such as the loss of life, extensive property damage, or a disastrous damage to the environment.

> **Example:** Some examples of safety critical embedded systems are: a flight-control system in an airplane, an electronic-stability program in an automobile, a train-control system, a nuclear reactor control system, medical devices such as heart pacemakers, the control of the electric power grid, or a control system of a robot that interacts with humans.

11.4.1 What Is Safety?

Safety can be defined as *the probability that a system will survive a given time-span without the occurrence of a critical failure mode that can lead to catastrophic consequences*. In the literature [Lal94] the magical number 10^9 h, i.e., 115,000 years, is the MTTF (mean-time-to-failure) that is associated with safety-critical operations. Since the hardware reliability of a VLSI component is less than 10^9 h, a safety-aware design must be based on hardware-fault masking by redundancy. It is impossible to achieve confidence in the correctness of the design to the level of the required MTTF in safety-critical applications by testing only – extensive testing can establish confidence in a MTTF in the order of 10^4 to 10^5 h [Lit93]. A formal reliability model must be developed in order to establish the required level of safety, considering the experimental failure rates of the subsystems and the redundant structure of the system.

Mixed-Criticality Architectures. Safety is a system property – the overall system design determines which subsystems are safety-relevant and which subsystems can fail without any serious consequences on the remaining safety margin. In the past, many safety-critical functions have been implemented on dedicated hardware, physically separated from the rest of the system. Under these circumstances, it is relatively easy to convince a certification authority that any unintended interference of safety-critical and non-safety-critical system functions is barred by design. However, as the number of interacting safety-critical functions grows, a sharing of communication and computational resources becomes inevitable. This results in a need of *mixed-criticality architectures*, where applications of different criticality can coexist in a single integrated architecture and the probability of any unintended interference, both in the domains of value and time, among these different-criticality applications must be excluded by architectural mechanisms. If mixed-criticality partitions are established by software on a single CPU, the *partitioning system software*, e.g., a hypervisor, is assigned the highest criticality level of any application software module that is executed on this system.

Fail-Safe Versus Fail-Operational. In Sect. 1.5.2 a *fail-safe system* has been defined as a system, where the application can be put into a *safe state* in case of a failure. At present, the majority of industrial systems that are safety-relevant fall into this category.

Example: In most scenarios, a robot is in a safe state when it ceases to move. A robot control system is safe if it either produces correct results (both in the domain of value and time) or no results at all, i.e., the robot comes to a standstill. The safety-requirement of a robot control system is thus a *high error-detection coverage* (see Sect. 6.1.2).

In a number of applications, there exists a basic mechanical or hydraulic control system that keeps the application in a safe state in case of a failure of the computer control system that optimizes the performance. In this case it is sufficient if the computer system is guaranteed to fail *cleanly* (see Sect 6.1.3), i.e., inhibits its outputs when a failure is detected.

Example: The ABS system in a car optimizes the braking action, depending on the surface condition of the road. If the ABS system fails cleanly, the conventional hydraulic brake system is still available to bring a car to a safe stop.

There exist safety-relevant embedded applications where the physical system requires the continuous computer control in order to maintain a safe state. A total loss of computer control may cause a catastrophic failure of the physical system. In such an application, which we call *fail-operational*, the computer must continue to provide an acceptable level of service, if failures occur within the computer system.

Example: In a modern airplane, there is no mechanical or hydraulic backup to the computer-based flight control system. Therefore the flight control system must be *fail-operational*.

Fail-operational systems require the implementation of active redundancy (as discussed in Sect. 6.4) to mask component failures.

In the future, it is expected that the number of *fail-operational systems* will increase for the following reasons:

1. The cost of providing two subsystems based on different technologies – a basic mechanical or hydraulic backup subsystem for basic safety functions and an elaborate computer-based control system to optimize the process – will become prohibitive. The aerospace industry has already demonstrated that it is possible to provide fault-tolerant computer systems that meet challenging safety requirements.
2. If the difference between the functional capabilities of the computer-based control system and the basic mechanical safety system increases further and the computer system is available most of the time, then the operator may not have any experience in controlling the process safely with the basic mechanical safety system any more.
3. In some advanced processes, computer-based non-linear control strategies are essential for the safe operation of a process. They cannot be implemented in a simple safety system any more.
4. The decreasing hardware costs make fail-operational (fault-tolerant) systems that require no expensive on-call maintenance competitive in an increasing number of applications.

11.4.2 Safety Analysis

The architecture of a safety-critical system must be carefully analyzed before it is put into operation in order to reduce the probability that an *accident* caused by a computer failure will occur.

Damage is a pecuniary measure for the loss in an accident, e.g., death, illness, injury, loss of property, or environmental harm. Undesirable conditions that have the potential to cause or contribute to an accident are called *hazards*. A hazard is thus a *dangerous state* that can lead to an accident, given certain environmental triggering conditions. Hazards have a *severity* and a *probability*. The severity is related to the worst potential damage that can result from the accident associated with the hazard. The severity of hazards is often classified in a severity class. The product of *hazard severity* and *hazard probability* is called *risk*. The goal of safety analysis and safety engineering is to identify hazards and to propose measures that eliminate or at least reduce the hazard or reduce the probability of a hazard turning into a catastrophe, i.e., to minimize the risk [Lev95]. A risk originating from a particular hazard should be reduced to a level that is *as low as reasonably practical* (*ALARP*). This is a rather imprecise statement that must be interpreted with good engineering judgment. An action that is provided to reduce the risk associated with a hazard to a tolerable level is called a *safety function*. *Functional safety* encompasses the analysis, design, and implementation of safety functions. There exists an international standard, IEC 61508 on *functional safety*.

Example: A risk minimization technique is the implementation of an independent safety monitor that detects a hazardous state of the controlled object and forces the controlled object into a safe state.

In the following we discuss two safety analysis techniques, *fault tree analysis* and *failure mode and effect analysis*.

Fault Tree Analysis. A *fault tree* provides graphical insight into the possible combinations of component failures that can lead to a particular system failure, i.e., an accident. Fault tree analysis is an accepted methodology to identify hazards and to increase the safety of complex systems [Xin08]. The fault tree analysis begins at the system level with the identification of the undesirable failure event (the *top event* of the fault tree). It then investigates the subsystem failure conditions that can lead to this top event and proceeds down the tree until the analysis stops at a basic failure, usually a component failure mode (events in ellipses). The parts of a fault tree that are still undeveloped are identified by the diamond symbol. The failure conditions can be connected by the AND or the OR symbol. AND connectors typically model redundancy or safety mechanisms.

Example: Figure 11.1 depicts the fault tree of an electric iron. The undesirable top event occurs if the user of the electric iron receives an electric shock. Two conditions must be satisfied for this event to happen: the metal parts of the iron must be under high voltage (hazardous state) and the user must be in direct or indirect contact with the metal parts, i.e., the user either touches the metal directly or touches a wet piece of cloth that conducts the

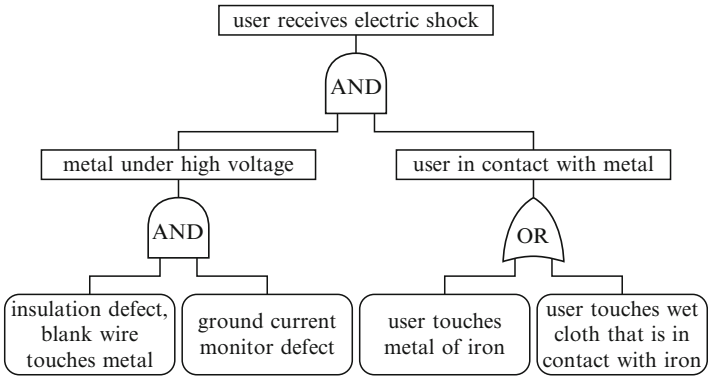

Fig. 11.1 Fault tree for an electric iron

electricity. The metal parts of the iron will be under high voltage if the insulation of a wire that touches the metal inside the iron is defect and the *ground-current monitor* that is supposed to detect the hazardous state (the metal parts are under high voltage) is defect.

Fault trees can be formally analyzed with mathematical techniques. Given the probability of basic component failures, the probability of the top event of a static fault tree can be calculated by standard combinatorial approaches.

Warm and cold spares, shared pools of resources, and sequence dependencies in which the order of the failure occurrence determines the state of the system, require more elaborate modeling techniques. A fault tree that cannot by analyzed by combinatorial approaches is called a *dynamic fault tree*. A dynamic fault tree is transformed into a Markov chain that can be solved by numerical techniques. There are excellent computer tools available that assist the design engineer in evaluating the reliability and safety of a given design, e.g., Mobius [Dea02].

Failure Mode and Effect Analysis (*FMEA*). Failure Mode and Effect Analysis (FMEA) is a bottom-up technique for systematically analyzing the effects of possible failure modes of components within a system to detect weak spots of the design and to prevent system failures from occurring. FMEA requires a team of experienced engineers to identify all possible failure modes of each component and to investigate the consequences of every failure on the service of the system at the system/user interface. The failure modes are entered into a standardized work sheet as sketched in Fig. 11.2.

A number of software tools have been developed to support the FMEA. The first efforts attempted to reduce the bookkeeping burden by introducing customized spreadsheet programs. Recent efforts have been directed towards assisting the reasoning process and to provide a system wide FMEA analysis [Sta03].

FMEA is complementary to the fault tree analysis, which was discussed in the previous section. While the fault tree analysis starts from the undesirable top event, and proceeds down to the component failures that are the cause of this system failure, the FMEA starts with the components and investigates the effects of the component failure on the system functions.

component	failure mode	failure effect	probability	criticality

Fig. 11.2 Worksheet for an FMEA

Table 11.1 Criticality level (Adapted from [ARI92])

Criticality	Failure of function
Level A	Results in catastrophic failure condition for the aircraft
Level B	Results in hazardous/severe-major failure condition for the aircraft
Level C	Results in major failure condition for the aircraft
Level D	Results in minor failure condition for the aircraft
Level E	Has no effect on aircraft operational capability or pilot workload

Dependability Modeling. A *dependability model* is a model of a distributed system constructed for the purpose of analyzing the *reliability of behavior* of the envisioned system. A good starting point for a reliability model is a structure block diagram derived from the architectural representation of the design, where the blocks are components and the connection among components are the dependencies among the components. The blocks are annotated with the failure rates and the repair rates of components, where the repair rate after a transient fault, closely related to the g-state cycle, is of particular importance, since most of the faults are transients. If there is any dependency among the failure rates of components, e.g., caused by the co-location of components on the same hardware unit, these dependencies must be carefully evaluated, since the correlated failures of components have a strong impact on the overall reliability. The correlation of failures among replicated components in a fault-tolerant design is of particular concern. There are a number of software tools to evaluate the reliability and availability of a design, such as the Mobius tool [Dea02].

The dependability analysis establishes the *criticality* of each function for the analyzed mission. The criticality determines the level of attention that must be given to the component that implements the function in the overall design of the system.

An example for the criticality level assignment of functions with respect to the airworthiness of a computer system onboard an aircraft is given in Table 11.1.

11.4.3 Safety Case

A *safety case* is a combination of a set of sound and well-documented arguments supported by analytical and experimental evidence concerning the safety of a

given design. The safety case must convince an independent certification authority that the system under consideration is safe to deploy. What exactly constitutes a proper safety case of a safety-critical computer system is a subject of intense debate.

Outline of the Safety Case. The safety case must argue why it is extremely unlikely that faults will cause a catastrophic failure. The arguments that are included in the safety case will have a major influence on design decisions at later stages of the project. Hence, the outline of the safety case should be planned during the early stages of a project.

At the core of the safety case is a rigorous analysis of the envisioned hazards and faults that could arise during the operation of the system and could cause catastrophic effects, such as harm to humans, economic loss, or severe environmental damage. The safety case must demonstrate that sufficient provisions (engineering and procedural) have been taken to reduce the risk to a level that is acceptable to society and why some other possible measures have been excluded (maybe due to economic or procedural reasons). The evidence is accumulated as the project proceeds. It consists of management evidence (ensuring that all prescribed procedures have been followed), design evidence (demonstrating that an established process model has been followed), and testing and operational evidence that is collected during the test phases and the operational phases of the target system or similar systems. The safety case is thus a *living document*.

A safety case will combine evidence from independent sources to convince the certification authority that the system is safe to deploy. Concerning the type of evidence presented in a safety case, it is commonly agreed that:

1. Deterministic evidence is preferred over probabilistic evidence (see Sect. 5.6).
2. Quantitative evidence is preferred over qualitative evidence.
3. Direct evidence is preferred over indirect evidence.
4. Product evidence is preferred over process evidence.

Computer systems can fail for external and internal reasons (refer to Sect. 6.1). External reasons are related to the operational environment (e.g., mechanical stress, external electromagnetic fields, temperature, wrong input), and to the system specification. The two main internal reasons for failure are:

1. The computer hardware fails because of a random physical fault. Section 6.4 presented a number of techniques how to detect and handle random hardware faults by redundancy. The effectiveness of these fault-tolerance mechanisms must be demonstrated as part of the safety case, e.g., by fault injection (Sect. 12.4).
2. The design, which consists of the software and hardware, contains residual design faults. The elimination of the design faults and the validation that a design (software and hardware) is *fit for purpose* is one of the great challenges of the scientific and engineering community. No single validation technology can provide the required evidence that a computer system will meet ultra-high dependability requirements.

Whereas standard fault-tolerance techniques, such as the replication of components for the implementation of triple-modular redundancy, are well established to mask the consequences of random hardware failures, there is no such standard technique known for the mitigation of errors in the design of software or hardware.

Properties of the Architecture. It is a common requirement of a safety critical application that no single fault, which is capable of causing a catastrophic failure, may exist in the whole system. This implies that for a *fail-safe application*, every critical error of the computer must be detected within such a short latency that the application can be forced into the safe state *before* the consequences of the error affect the system behavior. In a *fail-operational application*, a safe system service must be provided even *after* a single fault in any one of the components has occurred.

Fault-Containment Unit (*FCU*). At the architectural level, it must be demonstrated that *every* single fault can only affect a defined FCU and that it will be detected at the boundaries of this FCU. The partitioning of the system into independent FCUs is thus of utmost concern.

Experience has shown that there are a number of sensitive points in a design that can lead to a common-mode failure of all components within a distributed system:

1. A single source of time, such as a central clock.
2. A babbling component that disrupts the communication among the correct components in a communication system with shared resources (e.g., a bus system).
3. A single fault in the power supply or in the grounding system.
4. A single design error that is replicated when the same hardware or system software is used in all components.

Design Faults. A disciplined software-development process with inspections and design reviews reduces the number of design faults that are introduced into the software during initial development. Experimental evidence from testing, which in itself is infeasible to demonstrate the safety of the software in the ultra-dependable region, must be combined with structural arguments about the partitioning of the system into autonomous fault-containment units. The credibility can be further augmented by presenting results from formal analysis of critical properties and the experienced dependability of previous generations of similar systems. Experimental data about field-failure rates of critical components form the input to reliability models of the architecture to demonstrate that the system will mask random component failures with the required high probability. Finally, *diverse* mechanisms play an important role in reducing the probability of common-mode design failures.

Composable Safety Argument. Composability is another important architectural property and helps in designing a convincing safety case (see also Sect. 2.4.3). Assume that the components of a distributed system can be partitioned into two groups: one group of components that is involved in the implementation of safety critical functions and another group of components that is not involved in safety-critical functions. If it

can be shown at the architectural level, that no error in any one of the not-involved components can affect the proper operation of the components that implement the safety critical function, it is possible to exclude the not-involved components from further consideration during the safety case analysis.

11.4.4 Safety Standards

The increasing use of embedded computers in diverse safety-critical applications has prompted the appearance of many domain-specific safety-standards for the design of embedded systems. This is a topic of concern, since differing safety standards are roadblocks to the deployment of a cross-domain architecture and tools. A standardized unified approach to the design and certification of safety-critical computer system would alleviate this concern.

In the following, we discuss two safety standards that have achieved wide attention in the community and have been practically used in the design of safety-relevant embedded systems.

IEC 61508. In 1998, the International Electronic Commission (IEC) has developed a standard for the design of Electric/Electronic and Programmable Electronic (E/E/PE) safety related systems, known as *IEC 61508 standard* on *functional safety*. The standard is applicable to any safety-related control or protection system that uses computer technology. It covers all aspects in the software/hardware design and operation of safety-systems that operate on demand, also called *protection systems*, and safety-relevant control systems that operate in continuous mode.

Example: An example for a *safety system that operates on demand* (a protection system) is an *emergency shutdown system* in a nuclear power plant.

Example: An example of a *safety-relevant control system* is a control system in a chemical plant that keeps a continuous chemical process within safe process parameters.

The corner stone of *IEC 61508* is the accurate specification and design of the safety-functions that are needed to reduce the risk to a level *as low as reasonably practical* (*ALARP*) [Bro00]. The safety functions should be implemented in an independent safety channel. Within defined system boundaries, the safety functions are assigned to Safety-Integrity Levels (SIL), depending on the tolerated *probability for a failure on demand* for protection systems and a *probability of failure per hour* for safety-relevant control systems (Table 11.2).

The *IEC 61508* standard addresses random physical faults in the hardware, design faults in hardware and software, and failures of communication in a distributed system. IEC 61508-2 deals with the contribution of fault-tolerance to the dependability of the safety function. In order to reduce the probability of design faults of hardware and software, the standard recommends the adherence to a disciplined software development process and the provision of mechanisms that mitigate the consequences of remaining design faults during the operation of a system. It is interesting to note that dynamic reconfiguration mechanisms are not

Table 11.2 Safety integrity level (SIL) of safety functions

Safety integrity level	Average tolerated probability for a failure per demand	Average tolerated probability for a failure per hour
SIL 4	$\geq 10^{-5}$ to $<10^{-4}$	$\geq 10^{-9}$ to $<10^{-8}$
SIL 3	$\geq 10^{-4}$ to $<10^{-3}$	$\geq 10^{-8}$ to $<10^{-7}$
SIL 2	$\geq 10^{-3}$ to $<10^{-2}$	$\geq 10^{-7}$ to $<10^{-6}$
SIL 1	$\geq 10^{-2}$ to $<10^{-1}$	$\geq 10^{-6}$ to $<10^{-5}$

recommended in systems above SIL 1. IEC 61508 is the foundation for a number of domain specific safety standards, such as the emerging ISO 26262 standard for automotive applications, EN ISO 13849 for the machinery and off-highway industry, and IEC 60601 and IEC 62304 for medical devices.

Example: [Lie10] gives an example for the assignment of the *automotive safety integrity level* (ASIL) according to ISO 26262 to the two tasks, the *functional task* and the *monitoring task*, of an electronic actuator pedal (EGAS) implementation. If a *certified monitoring task* that detects an unsafe state and is guaranteed to bring the system into a safe state, is *independent* of the *functional task*, then the *functional task* does not have to be certified.

RTCA/DO-178B and DO-254. Over the past decades, safety-relevant computer systems have been deployed widely in the aircraft industry. This is the reason why the aircraft industry has extended experience in the design and operation of safety-relevant computer systems. The document *RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification* [ARI92] and the related document *RTCA/DO-254: Design Assurance Guidance for airborne electronic hardware* [ARI05] contain standards and recommendations for the design and validation of the software and hardware for airborne safety-relevant computer systems. These documents have been developed by a committee consisting of representatives of the major aerospace companies, airlines, and regulatory bodies and thus represent an international consensus view on a reasonable and practical approach that produces safe systems. Experienced with the use of this standard has been gained within a number of major projects, such as the application of *RTCA/DO-178B* in the design of the Boeing 777 aircraft and follow-on aircrafts.

The basic idea of *RTCA/DO-178B* is a two phase approach: in a first phase, the *planning phase*, the structure of the safety case, the procedures that must be followed in the execution of the project, and the produced documentation is defined. In the second phase, the *execution phase*, it is checked that all procedures that are established in the first phase are precisely adhered to in the execution of the project. The criticality of the software is derived from the criticality of the software-related function that has been identified during safety analysis and is classified according to Table 11.1. The rigor of the software development process increases with an increase in the criticality level of the software. The standard contains tables and checklists that suggest the design, validation, documentation, and project management methods that must be followed when developing software for a given criticality level. At higher criticality levels, the inspection procedures must be performed by personal that is independent from the development group. For the highest

criticality level, *level A*, the application of formal methods is recommended, but not demanded.

When it comes to the elimination of design faults, both standards, *IEC 61508* and *RTCA/DO-178B* demand a rigorous software development process, hoping that software is developed according to such a process will be free of design faults. From a certification point of view, an evaluation of the software product would be more appealing than an evaluation of the development process, but we must recognize there are fundamental limitations concerning the validation of a software product by testing [Lit93].

Recently, the new standard *RTCA/DO-297 Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations* has been published that addresses the role of design methodologies, architectures, and partitioning methods in the certification of modern integrated avionics systems in commercial aircraft. This standard also considers the contribution of time-triggered partitioning mechanisms in the design of safety-relevant distributed systems.

11.5 Design Diversity

Field data on the observed reliability of many large computer systems indicate that a significant and increasing number of computer system failures are caused by design errors in the software and not by physical faults of the hardware. While the problems of random physical hardware faults can be solved by applying redundancy (see Sect 6.4), no generally accepted procedure to deal with the problem of design (software) errors has emerged. The techniques that have been developed for handling hardware faults are not directly applicable to the field of software, because there is no physical process that causes the aging of the software.

Software errors are design errors that have their root in the unmanaged complexity of a design. In [Boe01] the most common software errors are analyzed. Because many hardware functions of a complex VLSI chip are implemented in microcode that is stored in a ROM, the possibility of a design error in the hardware must be considered in a safety-critical system. The issue of a single design error that is replicated in the software of all nodes of a distributed system warrants further consideration. It is conceivable that an FTU built from nodes based on the same hardware and using the same system software exhibits common-mode failures caused by design errors in the software or in the hardware (micro-programs).

11.5.1 Diverse Software Versions

The three major strategies to attack the problem of unreliable software are the following:

1. To improve the understandability of a software system by introducing a structure of conceptual integrity and by simplifying programming paradigms. This is, by

far, the most important strategy that has been widely supported throughout this book.
2. To apply formal methods in the software development process so that the specification can be expressed in a rigorous form. It is then possible to verify formally – within the limits of today's technology – the consistency between a high-level specification expressed in a formal specification language and the implementation.
3. To design and implement diverse versions of the software such that a safe level of service can be provided even in the presence of design faults.

In our opinion, these three strategies are not contradictory, but complementary. An understandable and well-structured software system is a prerequisite for the application of any of the other two techniques, i.e., program verification and software diversity. In safety-critical real-time systems, all three strategies should be followed to reduce the number of design errors to a level that is commensurate with the requirement of ultra-high dependability.

Design diversity is based on the hypothesis that different programmers using different programming languages and different development tools don't make the same programming errors. This hypothesis has been tested in a number of controlled experiments with a result that it is only partially encouraging [Avi85]. Design diversity increases the overall reliability of a system. It is, however, not justified to assume that the errors in the diverse software versions that are developed from the same specification are not correlated [Kni86].

The detailed analysis of field data of large software systems reveals that a significant number of system failures can be traced to flaws in the system specification. To be more effective, the diverse software versions should be based on different specifications. This complicates the design of the voting algorithm. Practical experience with non-exact voting schemes has not been encouraging [Lal94].

What place does software diversity have in safety critical real-time systems? The following case study of a fault-tolerant railway signaling system that is installed in a number of European train stations to increase the safety and reliability of the train service is a good example of the practical utility of software diversity.

11.5.2 An Example of a Fail-Safe System

The VOTRICS train signaling system that has been developed by Alcatel [Kan95] is an industrial example of the application of design diversity in a safety-critical real-time environment. The objective of a train signaling system is to collect data about the state of the tracks in train stations, i.e., the current positions and movements of the trains and the positions of the switches, and to set the signals and shift the switches such that the trains can move safely through the station according to the given timetable entered by the operator. The safe operation of the train system is of utmost concern.

The VOTRICS system is partitioned into two independent subsystems. The first subsystem accepts the commands from the station operators, collects the data from the tracks, and calculates the intended position of the switches and signals so that the train can move through the station according to the desired plan. This subsystem uses a TMR architecture to tolerate a single hardware fault.

The second subsystem, called the *safety bag*, monitors the safety of the state of the station. It has access to the real-time database and the intended output commands of the first subsystem. It dynamically evaluates safety predicates that are derived from the traditional "rule book" of the railway authority. In case it cannot dynamically verify the safety of an intended output state, it has the authority to block the outputs to the switching signals, or to even activate an emergency shutdown of the complete station, setting all signals to red and stopping all trains. The safety bag is also implemented on a TMR hardware architecture.

The interesting aspect about this architecture is the substantial independence of the two diverse software versions. The versions are derived from completely different specifications. Subsystem one takes the operational requirements as the starting point for the software specification, while subsystem two takes the established safety rules as its starting point. Common mode specification errors can thus be ruled out. The implementation is also substantially different. Subsystem one is built according to a standard programming paradigm, while subsystem two is based on expert-system technology. If the rule-based expert system does not come up with a positive answer within a pre-specified time interval, a violation of a safety condition is assumed. It is thus not necessary to analytically establish a WCET for the expert system (which would be very difficult).

The system has been operational in different railway stations over a number of years. No case has been reported where an unsafe state remained undetected. The independent safety verification by the safety bag also has a positive effect during the commission phase, because failures in subsystem one are immediately detected by subsystem two.

From this and other experiences we can derive the general principle that in a safety-critical system, the execution of every safety-critical function must be monitored by a second independent channel based on a diverse design. *There should not be any safety-critical function on a single channel system.*

11.5.3 Multilevel System

The technique described above can also be applied to fail-operational applications that are controlled by a two-level computer system (Fig. 11.3). The higher-level computer system provides full functionality, and has a high-error detection coverage. If the high-level computer system fails, an independent and differently designed lower-level computer system with reduced functionality takes over. The reduced functionality must be sufficient to guarantee safety.

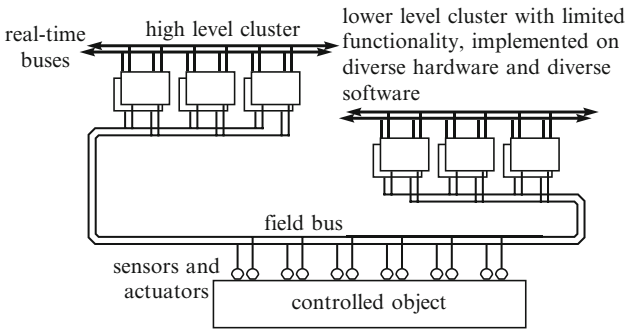

Fig. 11.3 Multilevel computer system with diverse software

Such an architecture has been deployed in the computer system for the space shuttle [Lee90, p. 297]. Along with a TMR system that uses identical software, a fourth computer with diverse software is provided in case of a design error that causes the correlated failure of the complete TMR system. Diversity is deployed in a number of existing safety critical real-time systems, as in the Airbus fly-by-wire system [Tra88], and in railway signaling [Kan95].

11.6 Design for Maintainability

The total cost of ownership of a product is not only the cost of the initial acquisition of the product, but the sum of the acquisition cost, the cost of operation, the expected maintenance cost over the product life, and finally, at the end of the product lifetime, the cost of product disposal. *Design for maintainability* tries to reduce the expected maintenance cost over the product lifetime. The cost of maintenance, which can be higher than the cost of the initial acquisition of the product, is strongly influenced by the product design and the maintenance strategy.

11.6.1 Cost of Maintenance

In order to be able to analyze the cost structure of a maintenance action, it is necessary to distinguish between two types of maintenance actions: *preventive maintenance* and *on-call maintenance*.

Preventive maintenance (sometimes also called *scheduled* or *routine maintenance*) refers to a maintenance action that is scheduled to take place periodically at planned intervals, when the plant or machine is intentionally shut down for maintenance. Based on knowledge about the increasing failure rate of components and the results of the analysis of the anomaly detection database (see Sect. 6.3), components that are

expected to fail in the near future are identified and replaced during preventive maintenance. An effective scheduled maintenance strategy needs extensive component instrumentation to be able to continually observe component parameters and learn about the imminent wear-out of components by statistical techniques.

On-call maintenance (sometimes also called *reactive maintenance*) refers to a maintenance action that is started after a product has failed to provide its service. By its nature it is unplanned. In addition to the *direct repair cost*, the on call-maintenance costs comprise the cost of *maintenance readiness* (to ensure the immediate availability of a repair team in case a failure occurs) and the cost of *unavailability of service* during the interval starting with the failure occurrence until the repair action has been completed. If an assembly line must be stopped during the *unavailability-of-service interval*, the cost of *unavailability of service* can be substantially higher than the initial product acquisition cost and the repair cost of the failed component.

Example: It is a goal of plant managers to reduce the probability for the need of on-called maintenance action as far as possible, ideally to zero.

Example: In the airline industry, unscheduled maintenance of an airplane means lost connections and extra cost for the lodging of passengers.

Another aspect that influences the cost of maintenance relates to the question whether *permanent hardware faults* or *software errors* are considered. The repair of a permanent hardware fault requires the physical replacement of the broken component, i.e., the spare part must be available at the site of failure and must be installed by a physical maintenance action. Given an appropriate infrastructure has been set up, the repair of a software fault can be performed remotely by downloading a new version of the software via the Internet with minimal or without any human intervention.

11.6.2 Maintenance Strategy

The design for maintenance starts with the specification of a maintenance strategy for a product. The maintenance strategy will depend on the *classification of components*, on the *maintainability*/*reliability*/*cost* tradeoff of the product, and the *expected use* of the product.

Component Classification. Two classes of components must be distinguished from the point of view of maintenance: components that exhibit *wear-out failures* and components that exhibit *spontaneous failures*. For components that exhibit wear-out failures, physical parameters must be identified that indicate the degree of wear-out. These parameters must be continually monitored in order to periodically establish the degree of wear-out and to determine whether a replacement of the component must be considered during the next scheduled maintenance interval.

Example: Monitoring the temperature or the vibration of a *bearing* can produce valuable information about the degree of wear-out of the bearing before it actually breaks down.

In some manufacturing plants, more than 100,000 sensors are installed to monitor wear out parameters of diverse physical components.

If it is not possible to identify a measureable wear-out parameter of a component or to measure such a parameter, another conservative technique of maintenance is the *derating of components* (i.e. operating the components in a domain where there is minimal stress on the components), and the systematic replacement of components after a given interval of deployment during a scheduled maintenance interval. This technique is, however, quite expensive.

For components with a spontaneous failure characteristic, such as many electronic components, it is not possible to estimate the interval that contains the instant of failure ahead of time. For these components the implementation of *fault-tolerance*, as discussed in Sect. 6.4, is the *technique of choice* to shift on-call maintenance to preventive maintenance.

Maintainability/Reliability Tradeoff. This tradeoff determines the design of the field-replaceable units (FRU) of a product. An FRU is a unit that can be replaced in the field in case of failure. Ideally, an FRU consists of one or more FCUs (see Sect. 6.1.1) in order that effective diagnosis of an FRU failure can be performed. The size (and cost) of an FRU (a *spare part*) is determined by a cost-analysis of a maintenance action on one side and the impact of the FCU structure on the reliability of the product on the other side. In order to reduce the time (and cost) of a repair action, the mechanical interfaces around an FRU should be easy to connect and disconnect. Mechanical interfaces that are easy to connect or disconnect (e.g., a plug) have a substantially higher failure rate than interfaces that are firmly connected (e.g., a solder connection). Thus, the introduction of FRU structure will normally *decrease* the product reliability. The most reliable product is one that cannot be maintained. Many consumer products fall into this category, since they are designed for optimal reliability – if the product is broken, it must be replaced as whole by a new product.

Expected Use. The expected use of a product determines whether a failure of the product will have serious consequences – such as the *downtime* of a large assembly line. In such a scenario it makes economic sense to implement a fault-tolerant electronic system that masks a spontaneous permanent failure of an electronic device. At the next scheduled maintenance interval, the broken device can be replaced, thus restoring the fault-tolerance capability. Hardware fault-tolerance thus transforms the expensive on-call maintenance action to a lower-cost scheduled maintenance action. The decreasing cost of electronic devices on one side and the increasing labor cost and the cost of production loss during on-call maintenance on the other side shift the break-even point for many electronic control systems towards fault-tolerant systems.

In an ambient intelligence environment, where smart Internet-enabled devices are placed in many homes, the maintenance strategy must ensure that non-experts can replace broken parts. This requires an elaborate diagnostic subsystem that diagnoses a fault to an FRU and orders the spare part autonomously via the Internet. If the spare part is delivered to the user, the inexperienced user must be capable to

replace the spare part with minimal effort in order to restore the fault-tolerance of the system with minimum mental and physical effort.

Example: The maintenance strategy of the *Apple iPhone* relies on the complete replacement of a broken hardware device, eliminating the need for setting up an elaborate hardware maintenance organization. Software errors are corrected semi-automatically by downloading a new version of the software from the *Apple iTunes store*.

11.6.3 Software Maintenance

The term *software maintenance* refers to all needed software activities to provide a useful service in a changing and evolving environment. These activities include:

- *Correction of software errors.* It is difficult to deliver error-free software. If dormant software errors are detected during operation in the field, the error must be corrected and a new software version must be delivered to the customer.
- *Elimination of vulnerabilities.* If a system is connected to the Internet, there is a high probability that any existing vulnerability will be detected by an intruder and used to attack and damage a system that would otherwise provide a reliable service.
- *Adaptation to evolving specifications.* A successful system changes its environment. The changed environment puts new demands on the system that must be fulfilled in order to keep the system relevant for its users.
- *Addition of new functions.* Over time, new useful system functions will be identified that should be included in a new version of the software.

The connection of an embedded system to the Internet is a mixed blessing. On one side, it makes it possible to provide Internet related services and to download a new version of the software remotely, but on the other side it enables an adversary to exploit vulnerabilities of a system that would be irrelevant if no Internet connection were provided.

Any embedded system that is connected to the Internet must support a *secure download service* [Obm09]. This service is absolutely essential for the continued remote maintenance of the software. The secure download must use strong cryptographic methods to ensure that an adversary cannot get control of the connected hardware device and download a software of its liking.

Example: A producer of modems sold 10,000 of modems all over the world before hackers found out that the modems contained a *vulnerability*. The producer did not consider to provide the infrastructure for a secure download service for installing a new corrected version of the software remotely.

Points to Remember

- In his recent book [Bro10], Fred Brook states that *conceptual integrity of a design is the result of a single mind.*

- Constraints limit the design space and help the designer to avoid the exploration of design alternatives that are unrealistic in the given environment. Constraints are thus our friends, not our adversaries.
- Software per se is an *action plan* describing the operations of a real or virtual machine. A plan by itself (without a machine) does not have *any temporal dimension*, cannot have *state* and has no *behavior*. This is one of the reasons why we consider the *component* and not the *job* as the primitive construct at the level of architecture design of an embedded system.
- *Purpose analysis*, i.e., the analysis why a new system is needed and what is the ultimate goal of a design must precede the requirements analysis.
- The analysis and understanding of a large problem is never complete and there are always good arguments for asking more questions concerning the requirements before starting with the *real* design work. The paraphrase *paralysis by analysis* has been coined to point out this danger.
- Model-based design is a design method that establishes a useful framework for the development and integration of executable models of the controlled object and of the controlling computer system.
- Component-based design is a *meet-in-the middle* design method. On the one side, the *functional and temporal requirements* on the components are derived top-down from the desired application functions. On the other side, the *functional and temporal capabilities* of the components are contained in the specifications of the available components.
- Safety can be defined as *the probability that a system will survive a given timespan without the occurrence of a critical failure mode that can lead to catastrophic consequences*.
- *Damage* is a pecuniary measure for the loss in an accident, e.g., death, illness, injury, loss of property, or environmental harm. Undesirable conditions that have the potential to cause or contribute to an accident are called *hazards*. A hazard is thus a *dangerous state* that can lead to an accident, given certain environmental triggering conditions.
- Hazards have a *severity* and a *probability*. The severity is related to the worst potential damage that can result from the accident associated with the hazard. The severity of hazards is often classified in a severity class. The product of *hazard severity* and *hazard probability* is called *risk*.
- The goal of safety analysis and safety engineering is to identify hazards and to propose measures that eliminate or at least reduce the hazard or reduce the probability of a hazard turning into a catastrophe, i.e., to minimize the risk.
- A *safety case* is a combination of a sound set of well-documented arguments supported by analytical and experimental evidence concerning the safety of a given design. The safety case must convince an independent certification authority that the system under consideration is safe to deploy.
- It is a goal of plant managers to reduce the probability for the need of on-called maintenance action as far as possible, ideally to zero.
- The connection of an embedded system to the Internet is a mixed blessing. On one side, it makes it possible to provide Internet related services and to download

a new version of the software remotely, but on the other side, it enables an adversary to exploit vulnerabilities of a system that would be irrelevant if no Internet connection were provided.

Bibliographic Notes

Many books have been written about design, most of them emanating from the field of architecture design. *Design Methods, Seeds of Human Futures* by Jones [Jon78] takes an interdisciplinary look at design and makes an enjoyable reading for a computer scientist, as well as the book *A Pattern Language* [Ale77] by Christopher Alexander. The excellent books *Systems Architecting, Creating and Building Complex Systems* [Rec91] and *The Art of System Architecting* [Rec02] by Eberhard Rechtin presents many empirically observed design guidelines that have been an important input for writing this chapter. The problem of software design for embedded systems is discussed in [Lee02]. The book *Embedded System Design* by Gajski [Gaj09] covers topics of hardware synthesis and verification.

Review Questions and Problems

11.1 Discuss the advantages and disadvantages of *grand design* versus *incremental development*.
11.2 Which are the characteristics of a "wicked" problem?
11.3 Why is the notion of a component, a hardware/software unit, introduced as the basic building block of a system? What are the problems with the notion of a software component in the context of real-time system design?
11.4 Discuss the different types of constraints that restrict a design. Why is it important to explore these constraints before starting a design project?
11.5 *Model-based design* and *component-based design* are two different design strategies. What are the differences?
11.6 What are the concepts behind UML MARTE and AADL?
11.7 Which are the results of the architecture design phase?
11.8 Establish a checklist for the evaluation of a design from the point of view of *functional coherence*, *testability*, *dependability*, *energy and power*, and *physical installation*.
11.9 What is a safety case? Which is the preferred evidence in a safety case?
11.10 Explain the safety-analysis techniques of Fault Tree Analysis and Failure Mode and Effect Analysis!
11.11 What is the key idea behind the safety standard IEC 6105?
11.12 What is a SIL?
11.13 What are the advantages and disadvantages of design diversity?
11.14 Discuss the reliability/maintainability tradeoff!
11.15 Why do we need to *maintain* software?
11.16 Why is a secure download service essential if an embedded system is connected to the Internet?