1

Real-time Characteristics and Safety of Embedded Systems

1.1 Introduction

What an embedded system is, is not exactly defined. In general, this term is understood to mean a special-purpose computer system designed to control or support the operation of a larger technical system (termed the embedding system) usually having mechanical components and in which the embedded system is encapsulated. Unlike a general-purpose computer, it only performs a few specific, more or less complex pre-defined tasks. It is expected to function without human interaction and, therefore, it usually has sensors and actuators, but not peripheral interfaces like keyboards or monitors, except if the latter are required to operate the embedding system. Often, it functions under real-time constraints, what means that service requests must be handled within pre-defined time intervals.

Embedded systems are composed of hardware and corresponding software parts. The complexity of the hardware ranges from very simple programmable chips (like field programmable gate arrays or FPGAs) over single microcontroller boards to complex distributed computer systems. Usually the software is stored in ROMs, as embedded systems seldom have mass storage facilities. Peripheral interfaces communicate with the process environments, and usually include digital and analogue inputs and outputs to connect with sensors and actuators.

In simpler cases, the software consists of a single program running in a loop, which is started on power-on, and which responds to certain events in the environment. In more complex cases, operating systems are employed, providing features like multitasking, different scheduling policies, synchronisation, resource management and others, to be dealt with later in this book.

The trend towards distributed architectures away from centralised ones assures modularity for structured design, better distribution of processing power, robustness, fault tolerance, and other advantages.

There are almost no areas of modern technology which could do without embedded systems. They appear in all areas of industrial applications and process control, in cars, in home appliances, entertainment electronics, cellular phones, video and photo cameras, and many more places. We even have them implanted, or wear them in our garments, shoes, or eye glasses. Their major spread has occurred particularly in the last decade. They pervade areas where they were only recently not considered. As they are becoming ubiquitous, we gradually do not notice them any more.

Contemporary cars, for example, contain dozens of embedded computers connected *via* hierarchically organised multi-level networks to communicate low-level sensory information or inter-processor messages, and to provide higher application-level interconnection of multimedia appliances, navigation systems, *etc.* The driver is not aware of the computers involved, but is merely utilising the new functionality. Within such automotive systems, there are also safety-critical components being prepared to deal in the near future with functions like drive-, brake-, or steer-by-wire. Should such a system fail, the consequences are much different than for, *e.g.*, Anti-Blocking Systems, whose function simply ceases in the case of a failure without putting users in immediate danger. With to the failure of an x-by-wire facility, however, drivers would not even be in a position to stop their cars safely.

Such considerations brought into light another aspect that has not been observed before. Since in the past embedded systems were considered to be sensitive high-technology elements, they were observed with a certain amount of precautious scepticism and doubt in their proper functioning. Special care was taken in their implementation, and they were not employed in the most safety-critical environments, like control units of nuclear power plants.

As a consequence of the increasing complexity of control algorithms in such applications, for better flexibility, and for economic reasons, however, and of getting used to their application in other areas, embedded systems have also found their way into more safety-critical areas where the integrity of the systems substantially depends on them. Any failures could have severe consequences: they may result in massive material losses or endanger human safety. Often the implementation of embedded systems is inadequate with regard to the means and/or the methods employed. Therefore, it is the prime goal of this book to point out what should be considered in the various design domains of embedded systems. A number of long existing guidelines, methods, and technologies of proper design will be mentioned and some elaborated in more detail.

By definition, embedded systems operate in the real-time domain, which means that their temporal behaviour is — at least — equally as important as their functional behaviour. This fact is often not considered seriously enough. There are a number of misconceptions that have been identified in an early paper by Stankovic [104]; some characteristic and still partially valid ones will be elaborated later in this chapter.

While verifying embedded systems' conformance to functional specifications is well established, temporal circumstances are seldom consistently verified. The methods and techniques employed are predominantly based on testing, and the quality achieved mainly depends on the experience and intuition of the designers. It is almost never proven at design time that such a system will meet its temporal requirements in every situation that it may encounter.

Unfortunately, this situation was identified more than 20 years ago, when the basic principles of the real-time research domain was already well organised. Although adequate partial solutions were known for a number of years, in practice embedded systems design did not progress essentially during this time. Therefore, in this work we present certain contributions to several critical areas of control systems design in a holistic manner, with the aim to improve both functional and temporal correctness. The implementation of long established, but often neglected, viable solutions will be shown with examples, rather than devising new methods and techniques. As verification of functional correctness is more established than that of temporal correctness, although equally important, special emphasis will be given to the latter.

While adequate verification of temporal and functional behaviour is important for high quality design of embedded systems, it cannot be taken as a sufficient basis to improve their dependability. It is necessary to consider the principles of fault management and safety measures for such systems in the early design phases, which means that common commercial off-the-shelf control computers are usually unsuitable for safety-critical applications.

In the late 1980s, the International Electrotechnical Commission (IEC) started the standardisation of safety issues in computer control [58]. It identified four Safety Integrity Levels (*SIL*), with SIL 4 being the most critical one (more details follow in Section 1.3.2). This book, however, is concerned with applications falling into the least demanding first level SIL 1, which allows the use of computer control systems based on generic microprocessors. It is desirable that such systems should formally be proven correct or even be safety-licensed. Owing to the complexity of software-based computer control systems, however, this is very difficult if not impossible to achieve.

1.2 Real-time Systems and their Properties

Let us start with some examples that demonstrate what real-time behaviour of a system actually is. A very good, but unexpected, example of proper and problem-oriented temporal behaviour, dynamic handling of priorities, synchronisation, adaptive scheduling and much more is the daily work of a housekeeper and parent, whose tasks are to care for children, to do a lot of housework and shopping, and to cook for the family. Apart from that, the housekeeper also receives telephone calls and visitors. Some of these tasks are known in advance and can be statically planned (scheduled), like sending children to school, doing laundry, cooking lunch, or shopping. On the other

hand, there are others that happen sporadically, like a visit of a postman, telephone calls, or other events, that cannot be planned in advance. The reactions to them must be scheduled dynamically, *i.e.,* current plans must be adapted when such events occur.

For statically scheduled tasks, often a chain of activities must be properly carried through. For instance, to send the children to the school bus, they must be woken on time, they must use the bathroom along with other family members, enough time must be allowed for breakfast which is prepared in parallel with the children being in the bathroom and getting dressed. The deadline is quite firm, namely, the departure of the school bus. In the planning, enough time must be allocated for all these activities. It is not a good idea, however, to allow for too much slack, since the children should not have to get up much earlier than necessary, thus losing sleep in the morning.

After sending the children to school, there are further tasks to be taken care of. Housekeeping, laundry, cooking and shopping are carried out in an interleaved manner and partly in parallel. Some of these tasks have more or less strict deadlines (*e.g.,* lunch should be ready for the children coming in from school). The deadlines can be set according to the time of the day (or the clock) or relative to the flow of other events. If the housekeeper is cooking eggs or boiling milk, the time until they will be ready is known in advance. If a sporadic event like a telephone call or postman's visit occurs during that time, the housekeeper must decide whether to accept it or not. If the event is urgent, it may be decided to re-schedule the procedure and interrupt cooking until the event is taken care of. Needless to say, that there are events with high and absolute priorities that will be handled regardless of other consequences; if, for example, a child is approaching a hot electric iron, then the housekeeper will interrupt any other activity whatsoever, even at the cost of milk boiling over.

Knowing his or her resources well, the housekeeper behaves very rationally. If, for instance, food provisions are kept in the same room where the laundry is done, the housekeeper will collect the vegetables needed for cooking when going there to start the washing machine, although they will not be needed until a later stage in the course of the housework planned, *e.g.,* after having made the beds.

1.2.1 Definitions, Classification and Properties

Following the pattern of the above example, in technical control systems there is usually a process that needs to be carried through. A *process* is the totality of activities in a system which influence each other and by which material, energy, or information is transformed, transported, or stored [28]. Specifically, a technical process is a process dealing with technical means. The basic element of a process is the *task*. It represents the elementary and atomic entity of parallel execution. The task concept is fundamental for asynchronous pro-

gramming. It is concerned with the execution of a program in a computing system during the lifetime of a process.

Considering the housewife example again, it is interesting that very complex sequences of tasks are quite normal for ordinary people like in the housekeeper example, and are carried out just with common sense. In the so-called "high technology" world of computers, however, people are reluctant to consider similar problems that way. Instead, sophisticated methods and procedures are devised to match obsolete approaches that were used in the past due to under-development, such as static priority scheduling.

Control systems should be considered in terms of tasks with their inherent natural properties. Each one's urgency is expressed by its deadline and not by artificially assigned priorities. This concept matches the natural behaviour of the housewife, as it is her goal to *perform her tasks in such a sequence and schedule that each tasks will be completed before its deadline.* This natural perception of tasks, priorities and deadlines is the essence of real-time behaviour:

> *In the real-time operating mode of a computer system the programs for the processing of data arriving from the outside are permanently ready, so that their results will be available within predetermined periods of time [27].*

Let us now consider two further examples that will lead us to a classification of real-time systems.

In preparation for a journey, we visit a travel agent to book a flight and buy tickets. The agent's job is to see which flights are available, to check the prices, and to make a reservation. If the service is busy, or there are any other unfortunate circumstances, this can take some time, or could even not be completed during our margin of patience. In the latter case, the agent could not fulfill the job, and we did not get our tickets. The deadline that has not been met was not very firmly set; it depended on a number of circumstances, *e.g.*, we were in a hurry or in a bad mood. Also, the longer we had to wait, the higher the probability that we would go to another agent next time.

When we go to the airport after the booking, the deadlines are set differently: if we are for some reason late and arrive after the door is closed (that deadline was known to us in advance), we have failed. It does not matter if we were late only by a few seconds or an hour. It does not even matter if we made any other functional mistake, for example went to wrong airport: it is the same if the failure to board was due to a functional or temporal error.

Considering the two examples above, we can classify the real-time systems into two general categories: systems with *hard and soft real-time behaviour.* Their main difference lies in the cost or penalty for missing their deadlines (see Figure 1.1). In the case of soft real-time systems, like in our example of flight ticketing, after a certain deadline the costs or penalty (customer dissatisfaction and, consequently, possibility of losing the customer) begin to rise. After a certain time, the action can be considered to have failed.

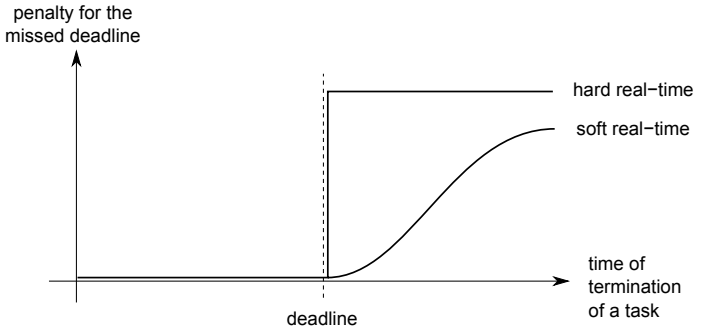

Fig. 1.1. Soft *vs* hard real-time temporal behavioural

In the case of hard real-time systems, as in our second example of missing a flight, the action has failed immediately after the deadline is missed. The cost or penalty function exhibits a jump to a certain high value indicating total failure, which may be high material costs or even endangering of environmental or human safety. Hence, hard real-time systems are those for which it holds that:

> *although functionally correct, the results produced after a certain predefined deadline are incorrect and, thus, useless.*

A task's failure to perform its function in due time may have different consequences. According to them, the hard- or soft real-time task may or may not be *mission-critical*, *i.e.*, the main objective of the entire application could not be met. Sometimes, however, a failure of a task can be tolerated, *e.g.*, when as a consequence only the performance is reduced to a certain extent. For instance, MPEG video-decoders in multimedia applications operate in the hard real-time mode: if a frame could not be decoded and composed before it would have to be put on screen, which is a hard deadline, the task failed as the frame is missing. The consequence would be flickering, which can be tolerated if it does not happen often — thus, it is not mission-critical.

On the other hand, soft real-time systems can be *safety-critical*. As an example, let us consider a diagnostics system whose goal is to report a situation of alert. Since human reaction times are relatively long and variable, it is not sensible to require the system's reaction to be within a precisely defined timeframe. However, the action's urgency increases with delay. The soft real-time deadline has a very positive side effect, namely, it allows other tasks more time to deal with the situation causing the alert and possibly to solve it.

Figure 1.1 depicts, and the definitions describe, two extreme cases of hard and soft real-time behaviour. In reality, however, the boundaries are often not so strict. Moreover, beside cost, benefit functions may also be considered, and different curves can be drawn [97]. Jensen describes the problem colourfully:

> "They (the real-time research community) have consensus on a precise technical (and correct) definition of "hard real-time," but left "soft real-time" to be tautologically defined as "not hard" — that is accurate and precise, but no more useful than dichotomising all colours into "black" and "not black" [67].

Together with Gouda and others [44] he has further elaborated the issue with "Time/Utility Functions" based on earliness, tardiness and lateness.

From the above we can conclude that *predictability of temporal behaviour* is the ultimate property of real-time systems. The necessary condition is determinism of temporal behaviour of the (sub-) systems. Strict and realistic predictability, however, is very difficult to achieve — practically impossible regarding the hardware and system architectures as employed in state-of-the-art embedded control systems. Hence, a much more pragmatic approach is needed.

In [105], Stankovic and Ramamritham elaborate two different approaches to predictability: the layer-by-layer (microscopic) and the top-layer (macroscopic) approach. The former stands for low-level predictability which is derived hierarchically: a layer in the design of a real-time system (processor, system architecture, scheduling, operating system, language, application) can only be predictable if all underlying layers are predictable. This type of predictability is necessary for low-level critical parts of real-time systems, and it should be provable.

For the higher layers (real-time databases, artificial intelligence, and other complex controls) microscopic predictability cannot be achieved. In these cases it is important that best effort is to be devoted, and that temporal behaviour is observed. The goal is to meet the deadlines in most cases. However, since it was not possible to prove that they are met in all cases, provisions should be made for the rare occasions of missed deadlines. Fault tolerance means should be implemented to resolve this situation. These must be simple and, thus, provably predictable in the microscopic sense.

The history of systematic research into real-time systems goes back at least to the 1970s. Although many solutions to the essential questions have been found very early, there are still many misconceptions that characterise this domain. In 1988, Stankovic collected many of them [104]. He found that one of the most characteristic misconceptions in the domain of hard real-time systems is that real-time computing is often considered as fast computing; probably to a lesser extent, this misconception is still alive. It is obvious from the above-mentioned facts that computer speed itself cannot guarantee that specified timing requirements will be met. Instead, predictability of temporal behaviour has been recognised as the ultimate objective. Being able to assure that a process will be serviced within a predefined timeframe is of utmost importance. Thus

A computer system can be used in real-time operating mode if it is possible to prove at design time that in all cases all requests will be served within predefined timeframes.

Beside timeliness, which is ensured by predictability, another requirement real-time systems should fulfill is *simultaneity*. This property is more severe, especially in multitasking and multiprocessor environments. It involves the demand that the execution behaviour of a process should be timely even in the presence of other parallel processes, whose number and behaviour are not known at design time and with whom it will share resources. It is not always possible to prove this property, but it should be considered and best efforts made.

Finally, real-time systems are inherently safety-related. For that reason, real-time systems should be dependable which, beside the properties of functional and temporal correctness, also includes robustness and permanent readiness. This property renders them particularly hard to design. The safety issues will be elaborated later in this chapter.

1.2.2 Problems in Adequate Implementation of Embedded Applications and General Guidelines

Although guidelines for proper design and implementation of embedded control systems operating in real-time environments have been known for a long time, in practice *ad hoc* approaches still prevail to a large extent. There are some major causes for this phenomenon:

- The basic problem seems to be the mismatch between the design objectives of generic universal computing and embedded control systems. It is reasonable to employ various low-level (caching, pipelining, *etc.*) and high-level measures (dynamic structures, objects, *etc.*) to achieve the best possible average performance with universal computers. Often, these measures are based on improvement of statistical properties and are, thus, in contradiction to the ultimate requirement of real-time systems, *viz.*, temporal determinism and predictability. There are no modern and powerful processors with easily predictable behaviour, nor compilers for languages that would prevent us from writing software with non-predictable run times. Practically all dynamic and "virtual" features aiming to enhance the *average* performance of non-real-time systems are, therefore, considered harmful. Inappropriate categories and optimality criteria widely employed in systems design are probabilistic and statistical terms, fairness in task processing, and minimisation of average reaction time. In contrast to this, the view adequate for real-time systems can be characterised by observation of hard timing constraints and worst cases, prevention of deadlocks, prevention of features taking arbitrarily long to execute, static analysis, and recognition of the constraints imposed by the real, *i.e.*, physical, world.

- The costs of consistently designed real-time embedded applications are much higher than conventional software. Timing circumstances need to be considered in all design stages, from specification to maintenance. Especially the verification and validation phases, when performed properly, are much more demanding and costly than in conventional computing.
- Designers of embedded systems are often reluctant to observe guidelines for proper design. Often overloaded, they tend to develop their applications in the usual way that was more or less appropriate in previous projects, but may fail in a critical situation. Owing to lack of time, knowledge, and will, they are not prepared to do the hard, annoying and time-consuming work of proving their designs' functional and temporal correctness.

The notion of time has long been ignored as a category in computer science. It is suggested in a natural way by the flow of occurrences in the world surrounding us. As the fourth dimension of our (Euclidean) space of experience, time is already a model defined by *law* and technically represented by Universal Time Co-ordinated (UTC). Time is an absolute measure and a practical tool allowing us to plan processes and future events easily and predictably with their mutual interactions requiring no further synchronisation. This is contrasted by the conceptual primitivity of computing, whose central notion algorithm is time-independent. Here, time is reduced to predecessor-successor relations, and is abstracted away even in parallel systems. No absolute time specifications are possible, the timing of actions is left implicit in real-time systems, and there are no time-based synchronisation schemes. As a result, the poor state of the "art" is characterised by computers using interval timers and software clocks with low (and in operation decreasing) accuracy, which are much more primitive than wrist watches. Moreover, meeting temporal conditions cannot be guaranteed, timer interrupts may be lost, every interrupt causes overhead, and clock synchronisation in distributed systems is still assumed to be a serious problem, although radio receivers for official date and time signals, as already available for 100 years and widely used for many purposes, providing the precise and worldwide only legal time UTC could easily and cheaply be incorporated in any node.

The core problem of contemporary information technology, however, is complexity, which is particularly severe in embedded systems design. It can be observed that people tend to use sophisticated and complicated measures and approaches when they feel that they need to provide good real-time solutions for demanding and critical applications. It is, however, much more appropriate to find simple solutions, which are transparent and understandable and, thus, safer. Simplicity is a means to realise dependability, which is the fundamental requirement of safety-related systems. (Easy) understandability is the most important precondition to prove the correctness of real-time systems, since safety-licensing (verification) is a social process with a legal quality.

There is a large number of examples for extensive complexity, or better, *"artificial complicatedness"*. Thus, for instance, the standard document

DIN 19245 of the fieldbus system Profibus consists of 750 pages, and a telephone exchange, which burned down in Reutlingen, had an installed software base of 12 million lines of code. On the other hand, a good example of successfully employing simple means in a high-technology environment is the general purpose computer used for the Space Shuttle's main control functions. It is based on five redundant IBM AP-101S computer systems whose development started in 1972; the last revision is from 1984, and it was deployed in 1991. They come out with 256k of 32 bit words of storage, and were programmed in the high-level assembly language HAL. Simplicity and stability of the design ensure the application's high integrity.

A serious problem in the design of safety-critical embedded systems is dependability of software:

> *We are now faced with a society in which the amount of software is doubling about every 18 months in consumer electronic devices, and in which software defect density is more or less unchanged in the last 20 years.*
>
> *In spite of this, we persist in the delusion that we can write software sufficiently well to justify its inclusion at the highest levels of safety criticality.*
>
> *Considering, for instance, the mean time between failure of a typical modern disk of around 500,000 h, the widening gulf between software quality and hardware quality becomes even more emphatic, to the point that the common procedure in safety critical systems of triplicating the same incredibly reliable hardware system and running the same much less reliable software in each channel seems questionable to say the least* [52].

Software must be valid and correct, which means that it must fulfil its problem specification. For the validity of specifications there is no more authority of control — except the developers' wishes, or more or less vaguely formulated requests. In principle, automatic verification is possible. Validation, on the other hand, is inherently hard, because it involves the human element to a great extent.

Software always contains design errors and, thus, needs correctness proofs, as tests cannot show the absence of errors. Safety-licensing of systems, whose behaviour is largely program-controlled, is still an unsolved problem, whose severity is increased by the legal requirement that verification must be based on object code. The still too big semantic gap between specifications on one hand and the too low a level programming constructs available on the other can be coped with by *the-other-way-around approach*, *viz.*, to select programming and verification methods of the utmost simplicity and, hence, highest trustworthiness, and to custom-tailor execution platforms for them.

Descartes (1641) pointed out the very nature of verification, which is neither a scientific nor a technical, but a *cognitive process*:

Verum est quod valde clare et distincte percipio.[1]

Verification is also a *social process*, since mathematical proofs rely on consensus between the members of the mathematical community. To verify safety-related computerised systems, this consensus ought to be as wide as possible. Furthermore, verification has a legal quality as well, in particular for embedded systems whose malfunctioning can result in liability suits. Simplicity can be used as the fundamental design principle to fight complexity and to create confidence. Based on simplicity, easy understandability of software verification methods — preferably also for non-experts — is the most important precondition to prove software correctness.

Design-integrated verification with the quality of mathematical rigour and oriented at the *comprehension capabilities of non-experts* ought to replace testing to facilitate safety-licensing. It should be characterised by simple, inherently safe programming — better specification, re-use of already licensed application-oriented modules, graphics instead of text, and rigourous — but not necessarily formal — verification methods understandable by non-experts such as judges. The more *safety-critical* a function is, the more *simple* the related software and its verification ought to be.

Simple solutions are the most difficult ones: they require high innovation and complete intellectual penetration of issues.

Progress is the road from the primitive via *the complicated to the simple.*

(Biedenkopf, 1994)

1.3 Safety of Embedded Computer Control Systems

To err is human, but to really foul things up requires a computer.

(Farmers' Almanac, 1978)

As society increasingly depends on computerised systems for control and automation functions in safety-critical applications and, for economical reasons, it is desirable to replace hardwired logic by programmable electronic systems in safety-related automation, there is a big demand for highly dependable programmable electronic systems for safety-critical embedded control and regulation applications. This domain forms a relatively new field, which still lacks its scientific foundations. Its significance arises from the growing awareness for safety in our society on the one hand, and from the technological trend towards more flexible, *i.e.*, program controlled, automation devices on the other hand. It is the aim to reach the state that computer-based systems can be constructed with a sufficient degree of confidence in their dependability.

[1] That which I perceive very clearly and distinctly is true.

Let us start with an example of a fault-tolerant design. In the Airbus 340 family, the fly-by-wire system, which is an extremely safety-critical feature, incorporates multiple redundancy [112]. There are three primary and two secondary main computers, each one comprising two units with different software. The primary and secondary computers run on different processors, and have different hardware and different architectures. They were designed and are supplied by different vendors. Only one flight computer is sufficient for full operation. Since mechanical signaling was retained for rudder movement and horizontal stabiliser trim, the aircraft can, if necessary, still be flown relying on mechanical systems only. Each computer has its command and monitoring units running in parallel; see Figure 1.2. They have separate hardware. The software for different channels in each computer was designed by different groups using different languages. Each control surface is controlled by different actuators which are driven by different computers. The hydraulic system is triplicated and the corresponding lines take different routes through the aircraft. The power supply sources and the signaling lanes are segregated.

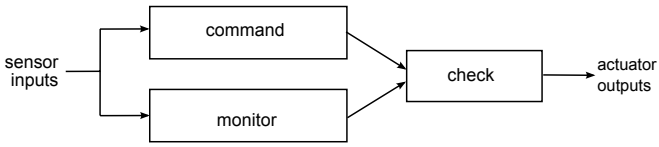

Fig. 1.2. Architecture of an A340 computer

In case of a loss of system resources, the flight control system may be reconfigured dynamically. This involves switching to alternative control software while maintaining system availability. Three operational modes are supported:

Normal - control plus reduction of workload,
Alternate - minimum computer-mediated control, and
Direct - no computer-mediation of pilot commands.

In spite of all these measures, there has been a number of incidents and accidents that may be related to the flight control system or its specifications, although a direct dependence has never been proven.

As functional and non-functional demands for computer systems have continued to grow over the last 30 years, so has the size of the resulting systems. They have become extremely large, consisting of many components, including distributed and parallel software, hardware, and communications, which increasingly interface with a large number of external devices, such as sensors and actuators. Another reason for large (and certain small) systems growing extensively complex is also the large number and complexity of interconnections between their components. Naturally, neither size nor number of connections nor components are the only sources of complexity. As users place increasing importance on such non-functional objectives as availability,

fault tolerance, security, safety, and traceability, the operation of a complex computer system is also required to be "non-stop", real-time, adaptable, and dependable, providing graceful degradation.

It is typical that such systems have lifetimes measured in decades. Over such periods, components evolve, logical and physical interconnections change, and interfaces and operational semantics do likewise, often leading to increased system complexity. Other factors that may also affect complexity are geographic distribution of processing and databases, interaction with humans, and unpredictability of system reactions to unexpected sequences of external events. When left unchecked, non-functional objectives, especially in legacy systems, can easily be violated. For instance, there are big, commercial off-the-shelf, embedded systems now running large amounts of software basically unknown to the user, which are problematic when trying to use them for real-time applications.

The safety of control systems needs to be established by certification. In that process, developers need to convince official bodies that all relevant hazards have been identified and dealt with. Certification methods and procedures used in different countries and in different industry domains vary to a large extent. Depending on national legislation and practice, currently the licensing authorities are still very reluctant or even refuse to approve safety-related technical systems, whose behaviour is exclusively program-controlled. In general, safety-licensing is denied for highly safety-critical systems relying on software with non-trivial complexity. The reasons lie mainly in a lack of confidence in complex software systems, and in the considerable effort needed for their safety validation. In practice, a number of established methods and guidelines have already proven its usefulness for the development of high integrity software employed for the control of safety-critical technical processes. Prior to its application, such software is further subjected to appropriate measures for its verification and validation.

However, according to the present state-of-the-art, all these measures cannot guarantee the correctness of larger programs with mathematical rigour. The method of diverse back-translation, for instance, which is the only general method approved by TÜV Rheinland (a public German licensing authority) to verify safety-critical software, is so cumbersome that up to two person-months are needed to verify just 4kB of machine code [48]. Practice has shown that even such small software components may include severe deficiencies as software developers mainly focus on functionality and often neglect safety issues. The problems encountered are exacerbated by the need to verify proper real-time behaviour.

1.3.1 Brief History of Safety Standards Relating to Computers in Control

This section provides a brief historical overview of the most important international, European and German safety standards. The list is roughly ordered by the year of publication.

DIN V VDE 0801 and DIN V 19250: [31, 32]
These documents belong to the first German safety standards applicable to general electric/electronic/programmable electronic (E/E/PE) safety-related systems comprehensively covering software aspects. Previous standards that dealt with the use of software covered only few life-cycle activities and were rather sector-specific, *e.g.*, IEC 60880 [57] "Software for Computers in the Safety Systems of Nuclear Power Stations". Although officially published in different years, *viz.*, DIN V VDE 0801 in 1990 and DIN V 19250 in 1994, there is a close link between them. They establish eight safety requirement classes (German: **A**nforderungs**k**lassen), with AK 1 the lowest and AK 8 the highest.

DIN V VDE 0801: Principles for using Computers Safety-related Systems
This standard defines techniques and measures required to meet each of the requirement classes. It includes techniques to control the EFFECT of hardware failures as well as measures to avoid the insertion of design-faults during hardware and software development. These measures cover design, coding, implementation, integration and validation, but the life-cycle approach is not explicitly mentioned.

DIN V 19250: Control Technology; Fundamental Safety Aspects for Measurement and Control Equipment
This standard specifies a methodology to establish the potential risk to individuals. The methodology takes the consequences of failures as well as the their probabilities into account. A risk graph is used to map the potential risk to one of the eight requirement classes.

EUROCAE-ED-12B: Software Considerations in Airborne Systems and Equipment Certification [38]
This standard, which is equivalent to the US standard RTCA DO-178B, was drafted by a co-operation of the European Organisation for Civil Aviation Equipment (EUROCAE) and its US counterpart Radio Technical Commission for Aeronautics (RTCA). It was released in 1992 and replaces earlier versions published in 1982 (DO-178/ED-12) and in 1985 (DO-178A/ED-12A). The standard considers the entire software life-cycle and provides a thorough basis for certifying software used in avionic systems like airplanes. It defines five levels of criticality, from A (Software whose failure would cause or contribute to a catastrophic failure of the aircraft) to E (Software whose failure would have no effect on the aircraft or on pilot workload).

EN 954: Safety of Machinery — Safety-related Parts of Control Systems [37]
This standard was developed by the European Committee for Standardisation (CEN) and has two parts: *General Principles for Design* and *Validation, Testing, Fault Lists*. Part 1 was first released in 1996, Part 2 in 1999. The standard complies with the basic terminology and methodology introduced in EN 292-1 (1991), and covers the following five steps of the safety life-cycle: hazard analysis and risk assessment, selection of measures to reduce risk, specification of safety requirements that safety-related parts must meet, design, and validation. It defines five safety categories: B, 1, 2, 3 and 4. The lowest category is B which requires no special measures for safety, and the highest is 4 requiring sophisticated techniques to avoid the consequences of any single fault. The standard focuses merely on the application of fault tolerance techniques in parts of machinery, it does not consider the system and its life-cycle as a whole [102].

ANSI/ISA S84.01: Application of Safety Instrumented Systems for the Process Industry [3]
This is the US standard for safety systems in the process industry. It was primarily introduced in 1996, and founded on the draft of IEC 61508 published in 1995. The standard follows nearly the same life-cycle approach as IEC 61508 and, thus, can be considered a sector-specific derivative of this umbrella standard. The specialisation on the process industry becomes apparent by its strong focus on *Safety Instrumented Systems* (SIS) and *Safety Instrumented Functions* (SIFs). According to the standard, SISs transfer a process to a safe state in case predefined conditions are violated, such as overruns of pressure or temperature limits. SIFs are the actions that a SIS carries out to achieve this. Since the committee initially thought that SIL 4 applications do not exist in the process industry, the first edition defined only three SILs, which are equivalent to SIL 1 to 3 of IEC 61508. However, the new release, ANSI/ISA S84.00.01-2004, includes the highest class SIL 4.

IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic (E/E/PE) Safety-related Systems [58]
The first draft of this standard was devised by IEC's Scientific Committee 65A and published in 1995 under the name "IEC 1508 Functional Safety: Safety-related Systems". After it gained wide publicity, a revised version was released in December 1998 as IEC 61508. This version comprises seven parts:

Part 1: General requirements
Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems
Part 3: Software requirements
Part 4: Definitions and abbreviations

Part 5: Examples of methods for the determination of safety integrity levels

Part 6: Guidance on the application of IEC 61508-2 and IEC 61508-3

Part 7: Overview of techniques and measures

The first four parts are normative, *i.e.*, they state definite requirements, whereas Parts 5 to 7 are informative, *i.e.*, they supplement the normative parts by offering guidance rather than stating requirements.

The standard defines four Safety Integrity Levels (SILs). SIL 1 is the lowest, SIL 4 the highest safety class. It is important to note that SILs are measures of the safety requirements of a given process; an individual product cannot carry a SIL rating. If a vendor claims a product to be certified for SIL 3, this means that it is certified for use in a SIL 3 environment [102].

The standard has a "generic" character, *i.e.*, it is intended as basis for writing sector- or application-specific standards. Nevertheless, if application-specific standards are not available, this umbrella standard can be used on its own.

In December 2001, CENELEC published a European version as EN 61508. It obliged all its member countries to implement this European version at national level by August 2002, and to withdraw conflicting national standards by August 2004. That is why DIN V VDE 0801 and DIN V 19250, as well as their extensions, were withdrawn at that date.

EN 50126, EN 50128 and EN 50129: CENELEC railway standards [34, 35, 36]

These three standards represent the backbone of the European safety licensing procedure for railway systems. They were developed by the Comité Européen de Normalisation Electrotechnique (CENELEC), the European Committee for Electrotechnical Standardisation in Brussels.

EN 50126: Railway Applications — The Specification and Demonstration of Dependability, Reliability, Availability, Maintainability and Safety (RAMS)

EN 50128: Railway Applications — Software for Railway Control and Protection Systems

EN 50129: Railway Applications — Safety-Related Electronic Systems for Signaling

This suite of standards, which is often referred to as the "CENELEC railway standards", was created with the intention to increase compatibility between rail systems throughout Europe and to allow mutual acceptance of approvals given by the different railway authorities. EN 50126 was published in 1999, whereas EN 50128 and EN 50129, which represent application-specific derivatives of IEC 61508 for railways, were released in 2002.

IEC 61511: Functional Safety: Safety Instrumented Systems for the Process Industry Sector [59]
This safety standard was first released in 2003, and represents a sector-specific implementation of IEC 61508 for the process industry. Thus, it covers the same safety life-cycle approach and re-iterates many definitions of its umbrella standard. Aspects that are of crucial importance for this application area, such as sensors und actuators, are treated in considerably higher detail. The standard consists of three parts named "Requirements", "Guidance to Support the Requirements", and "Hazard and Risk Assessment Techniques".
In September 2004, the IEC added a "Corrigendum" to the standard, and the ANSI adopted this version as new ANSI/ISA 84.00.01-2004 (IEC 61511 MOD). The US version is identical to IEC 61511 with one exception, a "grandfather clause" that preserves the validity of approvals for existing SISs.

IEC 61513: Nuclear Power Plants — Instrumentation and Control for Systems Important to Safety — General Requirements for Systems [60]
This sector-specific derivative of IEC 61508 for nuclear power plants was primarily released in 2002. Other safety standards for nuclear facilities like, *e.g.*, IEC 60880 were revised in conformity with IEC 61508.

There are many more safety standards related to Programmable Electronic Systems (PES), especially in the military area. This sometimes causes uncertainty in choosing the standard applicable for a given application, *e.g.*, EN 954-1 or IEC 61508 [41]. Moreover, if a system is used in several regions with different legal licensing authorities, *e.g.*, intercontinental aircraft, they may need to conform with multiple safety standards.

The overview presented in this section highlights the importance of IEC 61508. Its principles are internationally recognised as fundamental to modern safety management. Its life-cycle approach and holistic system view is applied in many modern safety standards — not only the ones that fall under the regulations of CENELEC.

1.3.2 Safety Integrity Levels

In the late 1980s, the IEC started the standardisation of safety issues in computer control [58]. They identified four Safety Integrity[2] Levels SIL 1 to SIL 4, with SIL 4 being the most critical one. In Table 1.1, applicable programming methods, language constructs, and verification methods are assigned to the safety integrity levels.

[2] Safety integrity is the likelihood of a safety-related system to perform the required safety functions satisfactorily under all stated conditions within a stated period of time [107].

Table 1.1. Safety integrity levels

Safety integrity level	Verification method	Language constructs	Typical programming method
SIL 4	Social consensus	Marking table entries	Cause-effect tables
SIL 3	Diverse back translation	Procedure calls	Function block diagrams with formally verified libraries
SIL 2	Symbolic execution, formal correctness proofs	Procedure call, assignment, case selection, iteration restricted loop	Language subsets enabling (formal) verification
SIL 1	All	Inherently safe ones, application oriented ones	Static language with safe constructs

For applications with highest safety-criticality falling into the SIL 4 group, one is not allowed to employ programming means such as we are used to. They can only be "programmed" using cause-effect tables (such as programming of simple PLA[3], PAL and similar programmable hardware devices), which are executed by hardware proven correct. The rows in cause-effect tables are associated with events, occurrence of which gives rise to Boolean preconditions. They can be verified by deriving the control functions from the rules read out from the tables stored in permanent memory and comparing them with the specifications. In Figure 1.3 a safety-critical fire fighting application is presented as a combination of cause-effect tables and functional block macros.

At SIL 3, programming of sequential software is already allowed, although only in a very limited form as interconnection of formally verified routines. No compilers may be used, because there are no formally proven correct compilers yet. A convenient way to interconnect routines utilises Function Block Diagrams as known from programmable logic controllers [56]. The suitable verification method is diverse back-translation: several inspectors take a program code from memory, disassemble it, and derive the control function. If they can all prove that it matches the specifications, a certificate can be issued [73]. This procedure is very demanding and can only be used in the case of pre-fabricated and formally proven correct software components.

[3] Programmable Logic Array.

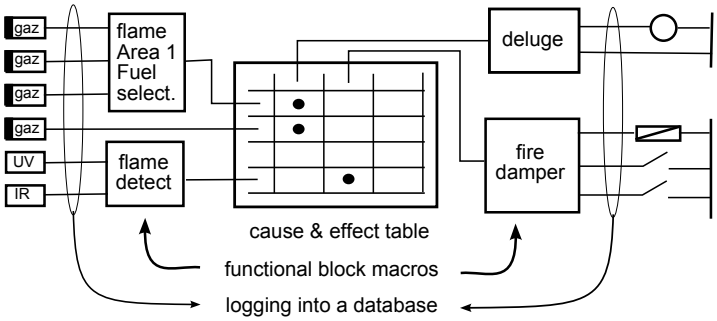

Fig. 1.3. An example of a safety-critical application

SIL 2 is the first level to allow for programming in the usual sense. Since formal verification of the programs is still required, only a safe subset of the chosen language may be used, providing for procedure calls, assignments, alternative selection, and loops with bounded numbers of iterations.

Conventional programming is possible for applications with the integrity requirements falling into SIL 1. However, since their safety is still critical, only static languages are permitted without dynamic features such as pointers or recursion that could jeopardise their integrity. Further, constructs that could lead to temporal or functional inconsistencies are also restricted. Any reasonable verification methods can be used.

In this book, applications falling into SIL 1 will be considered, although for safety back-up systems or partial implementations of critical subsystems higher levels could also apply. For that reason, in the sequel we shall only refer to SIL 1.

1.3.3 Dealing with Faults in Embedded Control Systems

A good systematic elaboration of handling faults and a taxonomy from this domain was presented by Storey [107]. Some points are summarised below. Faults may be characterised in different ways, for example, by:

Nature: random faults (hardware failure), systematic faults (design faults, software faults);
Duration: permanent (systematic faults), transient (alpha particle strikes on semiconductor memories), intermittent (faulty contacts); or by
Extent: local (single hardware or software module), global (system).

More and more, the general public is realising the inherent safety problems associated with computerised systems, and particularly with their software. Hardware is subject to wear, transient or random faults, and unintended environmental influences. These sources of non-dependability can, to a very large extent, be coped with successfully by applying a wide spectrum of redundancy and fault-tolerance methods.

Software, on the other hand, does not wear out nor can environmental circumstances cause software faults. Instead, software is imperfect, with all errors being design errors, *i.e.*, of systematic nature, and their causes always being latently present. They originate from insufficient insight into the problems at hand, leading to incomplete or inappropriate requirements and design flaws. Programming errors may add new failure modes that were not apparent at the requirements level. In general, not all errors contained in the resulting software can be detected by applying the methods prevailing in contemporary software development practice. Since the remaining errors may endanger the environment and even human lives, embedded systems are often less trustworthy than they ought to be. Taking the high and fast increasing complexity of control software into account, it is obvious that the problem of software dependability will exacerbate severely.

As already mentioned, due to the complexity of programmable control systems, faults are an unavoidable fact. A discipline coping with them is called "fault management". Broadly, its measures can be subdivided into four groups of techniques:

Fault avoidance aims to prevent faults from entering a system during the design stage,
Fault removal attempts to find faults before the system enters service (testing),
Fault detection aims to find faults in the system during service to minimise their effects, and
Fault tolerance allows the system to operate correctly in the presence of faults.

The best way to cope with faults is to prevent them from occurring. A good practice is to restrict the use of potentially dangerous features. Compliance with these restrictions must be checked by the compiler. For instance, dynamic features like recursion, references, virtual addressing, or dynamic file names and other parameters can be restricted, if they are not absolutely necessary.

It is important to consider the possible hazards, *i.e.*, the capability to do harm to people, property or the environment [107], during design time of a control system. In this sense the appropriate actions can be categorised as:

- Identification of possible hazards associated with the system and their classification,
- Determination of methods to dealing with these hazards,
- Assignment of appropriate reliability and availability requirements,
- Determination of an appropriate Safety Integrity Level, and
- Specification of appropriate development methods.

Hazard analysis presents a range of techniques that provide diverse insight into the characteristics of a system under investigation. The most common approaches are Failure Modes and Effects Analysis (FMEA), Hazard and Op-

erability Studies (HAZOP), and the Event- and Fault Tree Analyses (ETA and FTA).

Fault tree analysis in particular appears to be most suitable for use in the design of embedded control systems. It is a graphical method using symbols similar to those used in digital systems design, and some additional ones representing primary and secondary (the implicit) fault events to represent the logical function of the effects of faults in a system. The potential hazards are identified; then the faults and their interrelations that could lead to undesired events are explored. Once the fault tree is constructed it can be analysed, and eventually improvements proposed by adding redundant resources or alternative algorithms.

Since it is not possible in non-trivial cases to guarantee that there are no faults, it is important to detect them properly in order to deal with them. Some examples of fault-detection schemes are:

Functionality checking involves software routines that check the functionality of the hardware, usually memories, processor or communication resources.

Consistency checking. Using knowledge about the reasonable behaviour of signals or data, their validity may be checked. An example is range checking.

Checking pairs. In the case of redundant resources it is possible to check whether different instances of partial systems behave similarly.

Information redundancy. If feasible, it is reasonable to introduce certain redundancy in the data or signals in order to allow for fault detection, like checksums or parities.

Loop-back testing. In order to prevent faults of signal or data transmission, they can be transmitted back to the sources and verified.

Watchdog timers. To check the viability of a system, its response to a periodical signal is tested. If there is no response within a predefined interval, a timer detects a fault.

Bus monitoring. Operation of a computer system can often be monitored by observing the behaviour on its system bus to detect hardware failures.

It is advisable that these fault-detection techniques are implemented as operating system kernel functions, or in any other way built into the system software. Their employment is thus technically decoupled from their implementation allowing for their systematic use.

1.3.4 Fault-tolerance Measures

Approaches of Fault-Tolerant Control can be divided into two categories: passive and active fault tolerant control. The key difference between them is that an active fault tolerant control system includes a fault detection and isolation (FDI) system, and that fault handling is carried out based on information on faults delivered by the FDI system. In a passive fault tolerant control system,

on the other hand, the system components and controllers are designed to be robust to possible faults to a certain degree. Figure 1.4 sketches the basic classification of fault tolerant control concepts.

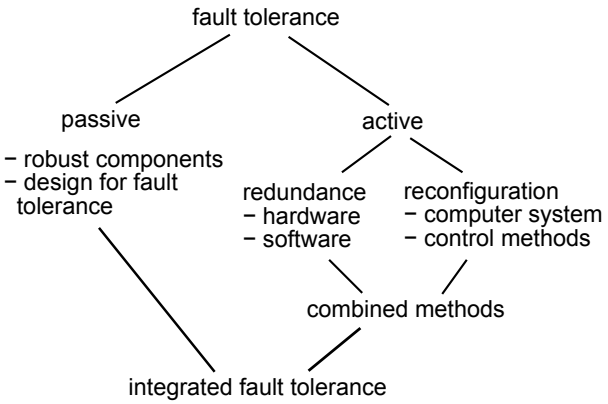

Fig. 1.4. Classification of fault-tolerance measures

Passive measures to improve fault tolerance mean that any reasonable effort must be made to make a design robust. For instance, the components must be selected accordingly, and with reasonable margins in critical features. Also, fault tolerance should already be considered in the design of subsystems. In addition to enhancing the quality and robustness of process components, using redundancy is a traditional way to improve process reliability and availability. However, because of the increased costs and complexity of the system, its usability is limited.

Evidently more flexible and cost effective is the reconfiguration scheme. Fault tolerance is achieved by system and/or controller reconfiguration, *i.e.*, after faults are identified and a reduction of system performance is observed, the overall system performance will be recovered (possibly to an acceptable degree, only) by a reconfiguration of parts of the control system under real-time conditions. This is a new challenge in the field of control engineering. In the following, the most common approaches for this are briefly sketched.

Redundancy

The most common measure to make a system tolerant to faults is to employ redundant resources. In the area of computing this idea originated in 1949: although still not tolerant to faults, EDVAC already had two ALUs to detect errors in calculation. Probably the first fault-tolerant computer was SAPO [87] built in Prague from 1950 to 1954 under the supervision of A. Svoboda, using relays and a magnetic drum memory. The processor used triplication and voting, and the memory implemented error detection with automatic retries

when an error was detected. A second machine developed by the same group (EPOS) also contained comprehensive fault-tolerance features.

The most simple model of redundancy-based fault tolerance is Triple Modular Redundancy (TMR): three resources with a voter allow for recognising a single resource failure. This is also called two-out-of-three (2*oo*3). The model can be extended to N-Modular Redundancy (NMR), which can accommodate more failures, but becomes more and more expensive. Seldom does N exceed 5 (3*oo*5). For other examples please refer to Section 7.3.1.

There are several ways in which the redundancy can be employed:

Hardware redundancy: e.g., in form of TMR,
Software redundancy: e.g., diversely implemented routines in recovery blocks, see below,
Information redundancy: e.g., as parity bits or redundant codes, and
Time redundancy: i.e., repetitive calculations to cope with the intermittent faults. fault-detection

Both in hardware and software redundancy it is most important to avoid common mode failures. They can be coped with successfully by design diversity. For a good example, see the description of the Airbus fly-by-wire computer control system on Page 14. To achieve a fully decoupled design, one should start with separate specifications to avoid errors in this stage, which are most costly and difficult to find by testing — a wrong system can thus even be verified formally!

Sometimes, the sources of common mode faults are deeply anchored in the designers by their education and training. This introduces a social component of fault management. This social problem of common mode failures was addressed by M.P. Hilsenkopf, an expert concerned with research and development for French nuclear power plants [53]. He pointed out that safety critical components are developed in parallel in two places by two different groups without a contact, *viz.*, in Toulouse and Paris. Starting with separate acquisition of requirements from physicists and engineers, they each provide their own specifications each. Based on that, they develop, verify and validate two different solutions. Then they swap them for final testing. Since they have both specified and developed their solutions, both groups know the problem very well, they know which questions and difficulties they had to solve, and they check how the respective competing group has coped with them. Thus, they verify each others' design on their own specifications, and both try to prove that their solution is better.

Reconfiguration

Owing to the fixed structure and high demands for hardware and software resources, and for economic reasons in less critical applications, the employment of the redundancy strategy is usually limited to certain specific technical processes or to key system components like the central computing system or

the bus system. To cover entire systems, fault-tolerant strategies with fault accommodation or system and/or controller reconfiguration are more suitable.

Owing to significantly different functions and working principles, the problems related to reconfiguration of control methods, algorithms and approaches as well as of the computer platforms, on which the latter and signal and data processing are running, are generally considered in separate and usually independent contexts. In this book we shall deal in more detail with the aspect of reconfiguring computer control systems, and with supporting the methods of higher-level control system reconfiguration.

When the occurrence of faults is detected, a system decides either to accommodate these faults or to perform controller and/or hardware reconfiguration, which might imply graceful performance degradation or, in the case of severe faults, to drive the component or process concerned into a safe mode. The decision on the type of action taken by the reconfiguration system is based on the evaluation of the actual system performance provided by a monitoring system, and on the need to assure an adequate transient behaviour upon reconfiguration.

Thus, for reconfiguration, designers prepare different solutions to a problem with gradually decreasing quality; the last one usually drives the system into a safe failing mode. For each of the solutions the necessary resources and the preconditions are listed. On the other hand, the computer system architecture provides for reconfiguration of the hardware in order to cope with resource failure. The optimum approach is normally run at the start, utilising all (or most of) the computer and control resources. This is also the main difference between redundancy and reconfiguration: most of the operational resources are always used.

Important components of reconfiguration-based fault-tolerant control are a fault detection system and a reconfiguration manager: based on the information provided by the former and on the resource needs attached to the gradually degraded options, the latter decides on the operating mode. Since the reconfiguration manager represents a central point of failure, it must be implemented in the safest possible way. It may be redundant or distributed among the other resources, which by a voting protocol then compete for taking over control. An example for the latter solution is the protocol to select a new time master in the case of failure in the TTCAN protocol [64]; *cf.* Section 3.5.2.

With respect to application design, dynamic reconfiguration resembles software redundancy (see below) — the recovery approaches or N versions of a control algorithm. According to the occurrence of faults, control will be switched to the most competent version, utilising the sound resources, and providing the best possible quality of service.

Hardware Fault Tolerance

In the design of fault-tolerant hardware, systems can be classified as static, dynamic or hybrid.

Static systems utilise fault effect masking, *i.e.*, preventing the effects of faults to propagate further into the systems. They use some voting mechanism to compare the outputs and mask the effect of faults. Well-known approaches include the n-modular redundancy, TMR with replicated voters (to eliminate the danger that a single voter fails).

Dynamic systems try to detect faults instead of masking their effects. The technique used is providing the stand-by resources. There is a single system producing output. If a fault is detected, control is switched to a redundant resource. The latter may have been running all the time in parallel with the master resource (hot stand-by), or was switched on when the master resource failed. In the case of hot redundancy, faults can be detected by, *e.g.*, self-checking pairs. In this case, obviously the redundant resource is more reliable, although probably delivering lower quality of service.

Static redundancy masks faults and they do not appear in the system. This is, however, achieved at a high cost for massive redundancy. Dynamic systems utilise less redundancy and provide continuous service, but introduce transient faults to the system: when an error occurs, the fault is in the first instant propagated, then coped with.

Hybrid systems combine the properties of the former two classes: by combining the techniques of dynamic redundancy with voters, they detect faults, mask their effects, and switch to stand-by units.

Software Fault Tolerance

Software fault tolerance copes with faults (of whatever origin) by software means rather than tolerating software errors. There are two major methods falling into this category, *N-version programming* and *recovery blocks*.

N-version programming resembles static hardware redundancy. A problem is solved by N different and diversely designed programs, all fulfilling the requirements stated in the specifications. All these programs are then run on the same data, either sequentially, or in parallel in the case of a multiprocessing system. If the results do not match, a fault is assumed and the results are blocked. The disadvantage of this method is that it either requires a multiprocessor system or takes more than N-times more time even if the system is working correctly.

Recovery blocks do not take much more time if there is no fault. Again, N versions of a program are designed, but only the most appropriate one is executed first. If a fault can be recognised, the results are discarded, the intermediate system states are reset, and the next alternative solution is tried. Eventually, one alternative must fulfil the requirements, or at least assure fail-safe behaviour. There are two different types of recovery blocks techniques [23, 24], backward and forward recovery.

The principle of *backward recovery* is to return to a previous consistent system state after an inconsistency (fault) is detected by consistency tests called

postconditions. This can be done in two ways; (1) by the operating system recording the current context before a program is "run", and restoring it after its unsuccessful termination, or (2) by recovery blocks inside the context of a task. The syntax of a recovery block *(RB)* could be

$$\text{RB} \equiv \textbf{ensure } \textit{post} \textbf{ by } \text{P}_0 \textbf{ else by } \text{P}_1 \textbf{ else by } \ldots \textbf{else } \textit{failure}$$

where P_0, P_1, *etc.* are alternatives which are consecutively tried until either consistency is ensured by meeting the *post*-condition, or the *failure*-handler is executed to bring the system into a safe state. Each alternative should independently be able to ensure consistent results.

The *forward error recovery technique* tries to obtain a consistent state from partly inconsistent data. Which data are usable can be determined by consistency tests, error presumptions, or with the help of independent external sources.

If in embedded process control systems an action directly affects a peripheral process, an irreversible change of initial states inside a failed alternative is caused. In this case, backward recovery is generally not possible. As a consequence, no physical control outputs should be generated inside the alternatives which may cause backward recovery in case of failure, *i.e.*, inside those which have postconditions. If this is not feasible, only forward recovery is possible, bringing the system into a certain predefined, safe, and stable state.

1.4 Summary of Chapter 1 and Synopsis of What Follows

In this introduction, some definitions have been laid down and the nature of real-time systems' execution behaviour has been demonstrated on examples. Safety issues have been discussed together with a brief enumeration of existing fault management measures and standards. This chapter has introduced terminology and provided the structure of guidelines on which the remaining chapters will build.

In the remainder of Part I, the concepts most important in designing distributed embedded real-time control systems will be elaborated. To start with, multitasking is the topic of Section 2, as it presents the nature of complex embedded control systems. In the section, first the approaches to task management are introduced and, then, two most common issues, scheduling and synchronisation, are dealt with. The preferred solutions are elaborated in more detail.

Sections 3 and 4 more specifically deal with hardware and software aspects of embedded systems design. They present some original solutions to certain problems, which have been developed by the authors, and which constitute guidelines for the implementation of platforms for distributed systems. In Part II the latter are elaborated in detail.