6

Design of Real-time Systems for QoS

It was discovered early, that in order to achieve the desired QoS properties a design methodology for real-time systems has to include appropriate measures to ensure that the QoS criteria are considered during the entire life-cycle. These have been joined in the framework of the ISO/IEC 13236 [31] and ISO/IEC TR 13243 [32] standards for QoS in information technology, and the standard IEC 61508 [28], which includes the necessary activities for safety-related systems from a project's start until the end of its life-cycle. Like safety, security is also an issue that is gaining importance for embedded (real-time) applications [76], and which must be dealt with during the design phase, too.

In the following sections the mentioned QoS criteria are addressed from different phases and aspects of real-time system design, where they should be considered in order for a system to fulfil these requirements. Apart from the general ISO/IEC 13236 and ISO/IEC TR 13243 standards, which mainly deal with system functionality and development process, more specific QoS standards for real-time systems are addressed here. Timeliness, being the most important QoS property of real-time systems, is considered in detail in the next chapter. It is, however, affected by the issues addressed here and cannot, therefore, be maintained in the long run unless they are also satisfied.

6.1 Design for Predictability and Dependability

6.1.1 Design for Predictability

For real-time systems the foremost property is predictability. It also pertains to behavioural predictability, which is addressed in the following sections. However, in the real-time domain usually a system's timeliness is meant, which represents the property of the system whether all its actions can be performed in time during its entire up-time. Such a system is considered to behave in a (temporally) predictable way, and is said to "operate in real time".

During design, predictability is supported by carefully planning the order of activities as well as taking care of their durations. Often the activities are executed periodically and, hence, a main loop, named *cyclic executive*, is built around them. It invokes them in a certain order when a timer, to which their period is assigned, times out. Naturally, the sum of their durations shall not exceed this period for them to finish in time. These activities, usually called *tasks*, do not necessarily have a unique and fixed period. Hence, dynamic scheduling algorithms have been devised (e.g. the Rate Monotonic one (RM)), which dynamically assign priorities to tasks based on their relative urgencies (reciprocal values of their periods). Also, some or all of the tasks may not be periodic, but still have temporal constraints on their execution. For these cases other non-priority- (time) oriented scheduling algorithms have been devised (e.g. Earliest Deadline First (EDF), or Least Laxity First (LLF)). They are considered in a special kind of performance analysis, named *schedulability analysis*, which determines the temporal predictability of diverse execution scenarios to discover bottlenecks and, foremost, to foresee the temporal predictability of a system's activities.

There are two kinds of design approaches that enable reasoning on the timeliness of task executions — formal and non-formal. The formal design methods encompass temporal state automata (e.g. Communicating Shared Resources (CSR) [69], timed Petri nets [79] or UML state charts). The non-formal design methods encompass Gantt diagrams and derivatives thereof (e.g. UML sequence diagrams and UML timing diagrams). While formal design methods can deliver estimates of an activities' execution time, it is still up to the scheduling algorithm to ensure their timely execution in correspondence with their periods/deadlines.

6.1.2 Design for Availability

Permanent readiness in high-availability systems requires that the systems be designed for non-stop operation. Of course, such systems also require maintenance and occasional software upgrades. For this purpose, re-configuration management mechanisms have been developed. Initially, static re-configuration has been used, representing strictly defined operation scenarios (e.g. manufacturing lines, space shuttle or avionics). With the advent of reactive systems the need for dynamic re-configuration arose, where the lines among phases are not so strict, and where there may be scenario parts that are interchanged leaving the rest intact and functioning.

Re-configuration management should support implementation platforms with the following attributes:

Heterogeneity: optimising architectures for performance, size and power consumption requires the most appropriate implementation techniques to be used; implementations require software-configurable hardware

(Hard) real time: usually there are significant real-time constraints on embedded systems

Re-configurability: an execution environment must allow hardware and software resources to be re-allocated dynamically

Execution environments supporting dynamic re-configuration encompass the following features:

- Specification of hardware and software configurations with well defined re-configuration scenarios — conditions and methods for re-allocation of hardware/software components and their interconnections
- Monitoring of local state changes and delegation of state changes to affected processing nodes during re-configuration
- Minimum and predictable overhead to overall execution time through short and well defined re-configuration actions

During dynamic re-configuration, application data must remain consistent and real-time constraints must be satisfied. To fulfil the mentioned requirements, these issues must be addressed at multiple levels:

Hardware level: At the lowest level, the hardware must be re-configurable. Software-programmable hardware components have best inherent hardware support, since their functions can be changed by memory contents. Also, internal hardware structures can be designed in a way restricting dangerous conditions that may damage hardware.

Middleware level: At a slightly higher level, the internal state of the system must be managed under changing tasking. Operating systems have evolved to support flexible implementations of multiple tasks on a single processor in the form of time-sharing and/or multitasking. Typically, however, this alone is not enough for low-level efficiency and/or hard real-time conditions.

Software level: To represent the behaviour of an embedded real-time system, generally three viewpoints must be considered: (1) the external functional viewpoint, representing the operation scenarios, (2) the internal functional viewpoint, which models the dynamical state changes of the system, and (3) the specification of the hardware and software architectures together with the mapping of software onto hardware components. These are the data required by the re-configuration management execution environment.

Re-configuration management for embedded (real-time) systems has mostly been dealt with in connection with their hardware/software co-design [47, 63, 78]. Besides defining diverse (dynamic) operation scenarios, two main goals have been achieved by this approach: (1) achieving fault tolerance by system design [25, 40], and (2) fast scenario switching for industrial automation and telecommunication systems [19, 27, 39, 63].

6.1.3 Design for Safety

In the late 1980s, the International Electrotechnical Commission (IEC) started the standardisation of safety issues in computer control. Four Safety Integrity Levels (SIL) were defined, with SIL4 being the most critical one. Activities were prescribed at different levels and phases of system development (e.g. coding standards, dynamic analysis and testing, black-box testing, failure analysis, modelling, performance testing, formal methods, static analysis, modular approach), which are desired or mandatory, and approaches that are allowed or required in order to fulfil the requirements of a certain Safety Integrity Level. These rules form the standard IEC 61508 for the life-cycle management of instrumented protection systems. As can be seen from Fig. 6.1, the safety life-cycle encompasses the entire production cycle from a system's design to its decommissioning.

SIL Flowchart

The flowchart in Fig. 6.1 represents the safety life-cycle of an Equipment Under Control (EUC) as a whole in its entirety. Such EUC is composed of one or more Electrical/Electronic/Programmable Electronic (E/E/PE) devices, which as a system have to fulfil individual as well as collective safety requirements. The single steps of the safety life-cycle for an EUC have the following meanings:

1. Concept: all information on the EUC (physical, legislative, etc.), relevant to the following steps need to be assembled
2. Overall scope definition: specification of the hazards and risks (e.g. process hazards, environmental hazards)
3. Hazard and risk analysis: is performed to determine the hazards and hazardous events or sequences of events of the EUC (and the EUC control system) for all foreseeable cases including fault conditions and misuse and to determine the associated risks
4. Overall safety requirements: a specification of the overall safety requirements is developed in terms of the safety function requirements and the safety integrity requirements in order to achieve the functional safety required
5. Safety requirements allocation: places the safety functions from the specification produced previously on the designated E/E/PE or on other-technology safety-related systems as well as external risk reduction facilities, and allocates a safety integrity level to each safety function
6. Overall operation and maintenance planning: is carried out for the E/E/PE safety-related systems to ensure that the required functional safety is maintained during operation and maintenance
7. Overall safety validation planning: facilitates the overall safety validation of the E/E/PE safety-related systems
8. Overall installation and commissioning planning: is carried out to ensure that during these phases the required functional safety of the E/E/PE safety-related systems is achieved
9. Safety-related systems (E/E/PES) realisation: comprises the creation phase of the E/E/PE safety-related systems where the specified functional safety and safety integrity requirements have to be obeyed
10. Safety-related systems (other technology) realisation: comprises the creation phase of other safety-related systems where the specified functional safety and safety integrity requirements for these specific technologies have to be obeyed (outside the scope of this standard)

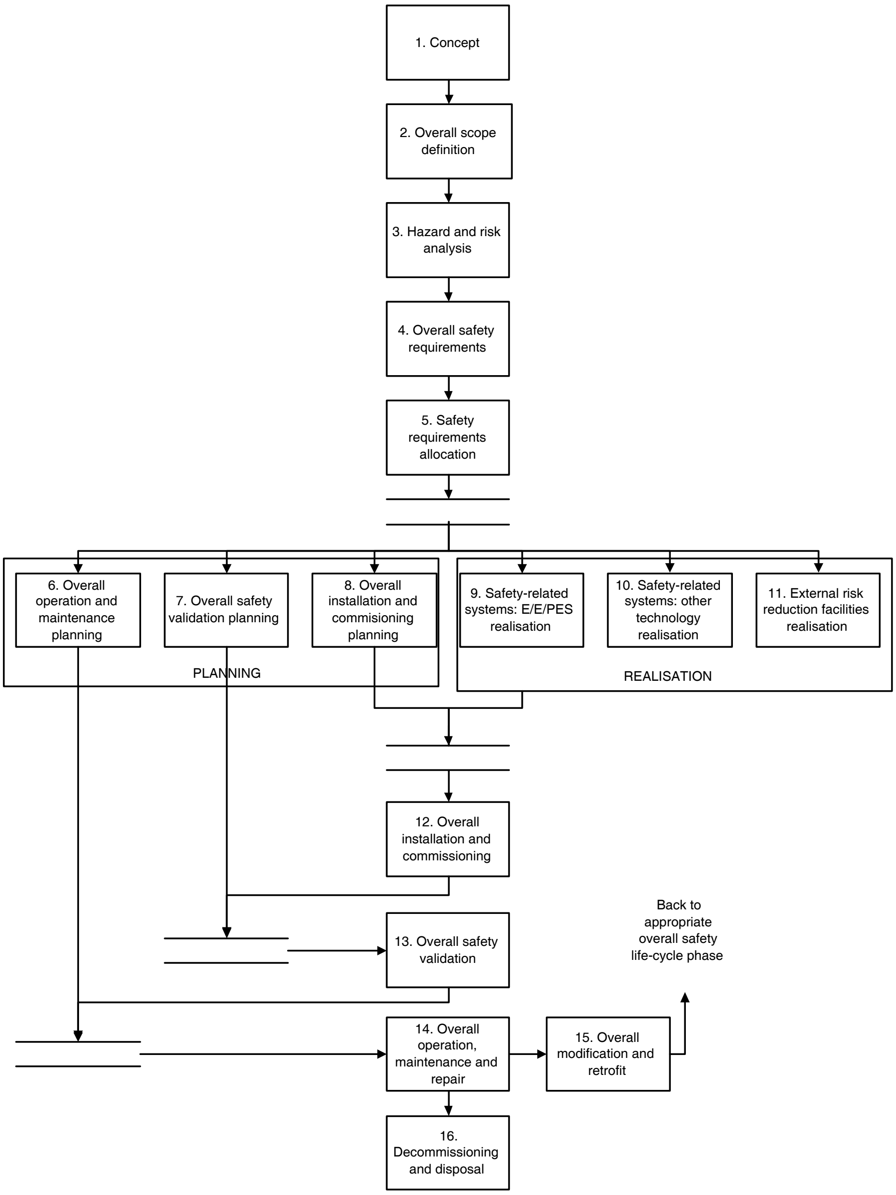
1. Concept
2. Overall scope definition
3. Hazard and risk analysis
4. Overall safety requirements
5. Safety requirements allocation
6. Overall operation and maintenance planning
7. Overall safety validation planning
8. Overall installation and commisioning planning
PLANNING
9. Safety-related systems: E/E/PES realisation
10. Safety-related systems: other technology realisation
11. External risk reduction facilities realisation
REALISATION
12. Overall installation and commissioning
13. Overall safety validation
Back to appropriate overall safety life-cycle phase
14. Overall operation, maintenance and repair
15. Overall modification and retrofit
16. Decommissioning and disposal

Fig. 6.1. Safety life-cycle according to IEC 61508

11. External risk reduction facilities realisation: comprises the creation of external risk reduction facilities to meet the specified safety functions and the safety integrity requirements thereof (outside the scope of this standard)
12. Overall installation and commissioning: comprises the installation/commissioning phase of the E/E/PE safety-related systems
13. Overall safety validation: is to validate that the specified functional safety and safety integrity requirements for the E/E/PE safety-related systems are met
14. Overall operation, maintenance and repair: comprises the with respect to safety functionally intact operation, maintenance and repair of the E/E/PE safety-related systems
15. Overall modification and retrofit: is meant to ensure that the functional safety for the E/E/PE safety-related systems is appropriate during and after these phases
16. Decommissioning or disposal: is meant to ensure that the functional safety of the E/E/PE safety-related systems is appropriate during and after these phases on the EUC and its control system

Safety Integrity Levels

The standard IEC 61508 details the requirements necessary for a system to qualify for each of the four Safety Integrity Levels. These requirements are more rigorous at higher levels of safety integrity in order to achieve the required lower likelihood of dangerous failures.

An E/E/PE safety-related system usually implements more than one safety function. If the safety integrity requirements for these safety functions differ, unless there is sufficient independence between their respective implementations, the requirements applicable to the highest relevant Safety Integrity Level shall apply to the entire E/E/PE safety-related system. If a single E/E/PE system is capable of providing all required safety functions, and the required safety integrity is less than that specified for SIL1, then IEC 61508 does not apply.

For safety-related systems two kinds of requirements are defined:

1. *Safety function requirements:* the safety functions that have to be performed
2. *Safety integrity requirements:* the reliability with which the safety functions have to be performed

By *functional safety* the ability of a safety-related system to carry out the actions necessary to achieve a safe state for the equipment under control or to maintain a safe state for the equipment under control is meant and relates to *safety*. *Safety integrity* is the likelihood of a safety-related system to achieve safety functions required under all conditions stated within a given period of time; it relates to *reliability*.

The Safety Integrity Levels are defined in Table 6.1 by the probabilities of "failure on demand (FOD)", "protective system technology", and "protective system design/testing/maintenance (D/T/M) requirements". We can observe how the increasing *complexity* of designs and components used also increases the cost and maintenance interval rate of systems. With falling maximum failure probability also the need to allow for common-cause failures is raised, which adds to complexity and cost. On the other hand, *simplicity* in design and usage of standard components with lower sensing and actuation (S&A) diversity prolong the required maintenance intervals and lower the need to allow for common-cause failures. Failure probability is the direct opposite of *dependability*.

Table 6.1. Safety Integrity Levels

FOD probability	
SIL1	0.1–0.01
SIL2	0.01–0.001
SIL3	0.001–0.0001
SIL4	0.0001–0.00001
System technology	
SIL1	Standard components, single channel
SIL2	Predominantly standard components
SIL3	Multiple channel with S&A diversity
SIL4	Specialised designs
D/T/M requirements	
SIL1	Relatively inexpensive
SIL2	Moderately expensive
SIL3	Expensive
SIL4	Very expensive
Test interval	
SIL1	$>$ 3 months
SIL2	$<$ 3 months
SIL3	1 month
SIL4	$<$ 1 month

Apart from the above-mentioned process techniques to achieve system safety, some design techniques have also been devised. The mentioned design techniques, representing vital parts of a system's development phase, form parts 6 and 9 of the safety life-cycle in Fig. 6.1. Some of them are summarised in Table 6.2 together with their importance to the individual *Safety Integrity Levels*.

We can observe similarities in recommended design techniques among SIL1 and SIL2, and among SIL3 and SIL4. The main differences are in the rigour of their application. There is a noticeable rise in the required effort and rigour of the methods to be applied between SIL2 and SIL3 (e.g. the use of formal methods).

In the following sections some of the mentioned design techniques are discussed and some further ones, which have been devised especially for fault removal or fault tolerance in dependable real-time systems, are presented.

Table 6.2. Software practices from IEC 61508-3 by category

Practice	61508-3	SIL1	SIL2	SIL3	SIL4
Coding standards					
Use of coding standard	B.1	HR	HR	HR	HR
No dynamically allocated variables	B.1	—	R	HR	HR
Dynamic analysis and testing					
Test case execution from cause consequence diagrams	B.2	—	—	R	R
Structure-based testing	B.2	R	R	HR	HR
Black-box testing					
Equivalence classes and input partition testing	B.3	R	HR	HR	HR
Failure analysis					
Failure modes, effects and criticality analysis	B.4	R	R	HR	HR
Formal methods modelling	B.5	—	R	R	HR
Performance modelling	B.5	R	HR	HR	HR
Timed Petri nets	B.5	—	R	HR	HR
Performance testing					
Avalanche/stress testing	B.6	R	R	HR	HR
Response timings and memory constraints	B.6	HR	HR	HR	HR
Performance requirements	B.6	HR	HR	HR	HR
Semi-formal methods					
Sequence diagrams	B.7	R	R	HR	HR
Finite state machines/state transition diagrams	B.7	R	R	HR	HR
Decision/truth tables	B.7	R	R	HR	HR
Static analysis					
Boundary value analysis	B.8	R	R	HR	HR
Control flow analysis	B.8	R	HR	HR	HR
Fagan inspections	B.8	—	R	R	HR
Symbolic execution	B.8	R	R	HR	HR
Walk-throughs/design reviews	B.8	HR	HR	HR	HR
Modular approach					
Software module size limit	B.9	HR	HR	HR	HR
Information hiding/encapsulation	B.9	R	HR	HR	HR
Fully defined interface	B.9	HR	HR	HR	HR
Total recommended (R)		12	12	3	1
Total highly recommended (HR)		6	10	20	22

Legend: HR highly recommended; R recommended; — no recommendation

6.1.4 Design for Reliability

Design for Testability

Maintainability and testability concern systems on which it is possible to act in order (1) to avoid the introduction of faults and, (2) when faults occur, to detect, localise and correct them. The means to tackle these issues can be divided into the following categories [21]:

Fault prevention aims to reduce the creation and occurrence of faults during the life-cycle of a system. It includes the measures to ensure that all *applicable physical constraints are met* and that *only static and real features are used*. As a precaution, to prevent faults in the design phase as well as to ensure maintainability, the concern for bounded complexity of the design as a whole and of its parts has to be taken into account as well. This eases portability and flexibility of a solution.

Fault removal aims to detect and eliminate existing faults. It is addressed by verification and validation of a system against its specifications.

Fault tolerance aims to guarantee the services provided by a system despite the presence or occurrence of faults. This issue is meant to ensure permanent readiness, simultaneous operation, predictability, robustness, graceful degradation and prevention of deadlocks.

Fault forecasting aims to estimate the presence of faults (number and severity).

Apart from the suppression of faults in real-time systems, the equally important timeliness issue has to be considered. In real-time systems, timing faults and functional faults are treated equally. Since these faults cannot be removed, they can only be addressed by fault forecasting and prevention, and handled by fault tolerance (e.g. graceful degradation) mechanisms.

To analyse the timing behaviour in a system, hardware and software monitors can be applied, and timing analysers used on the compiled code to estimate worst-case response times, error detection and correction times and signal-to-noise ratio bounds in signal processing applications. The quantities TBF, MTTF and MTTR are hard to measure, and exhaustive systematic testing has to be applied on a final system, possibly permanently damaging or destroying it.

Fault Prevention

Once the elicitation of a client's needs has been completed, a system is to be defined that fulfils these needs. This work leads to the expression of *specifications*. Since they are the first source of faults, they have to be well pondered and precisely formulated. To reduce the possibility of faults at this phase in the design process, the specifications may be checked for conformity with the following criteria:

1. *Semantic criteria*
   Non-ambiguity: each element of the specification should have one interpretation only
   Completeness: all aspects of operation should be covered and boundary conditions defined
   Consistency: there should be no conflicts between the elements of the specification.
   Traceability: the customer needs and specification elements have to be clearly correlated
2. *Syntactic criteria*
   Concision: unnecessary and misleading terms should be avoided
   Clarity: the sentences of the specification should be easy to read, i.e. short and simply formed; the text flow should be straightforward, i.e. unnecessary references and jumps should be avoided
   Simplicity: the concepts addressed have to be simple, their number has to be limited and they should be loosely coupled
   Comprehension: reading of the text has to facilitate the understanding of the semantics

Non-conformity to these criteria increases the risk of faults. These criteria also apply to design models. Methods to verify specifications comprise reviewing, scenarios and prototyping.

Review processes are carried out by humans analysing the contents of specifications. There are two general approaches to reviews: walk-through and inspection. A reviewer may search for faults or risks of faults in a specification model. More reliable results are produced if a review is conducted by a person or team that was not involved in the specification. Resulting notes on the (potential) faults are returned to the specification's producer.

From *scenarios* input/output sequences that simulate the interaction between the system specified and its environment are derived. They are presented to the client, who approves or disapproves them.

The result of *prototyping* is a tool, built from the specification document, which simulates the system's interactions with its environment. Its use by the client allows the detection of errors or misconceptions in the specification. Herewith, also possible sources of timing faults can be detected and dealt with appropriately.

The same guiding principles as for specifications remain pertinent also when choosing an appropriate design model. Methods of adequate expressiveness have to be selected in concordance with the characteristics of the system designed. The more the concepts to be modelled concur with the model features, the *simpler* the solution will be, and the lesser will be the probability of introducing faults. Versatile design methods for the hardware and software constituents of systems have already been partly unified (e.g. in the UML methodology dealt with in detail in the next chapter). There are, however, still aspects of hardware (e.g. boards) and software (e.g. temporal behaviour and analysis) design that are only partly addressed by the more general design methods. Hence, the pertaining properties have to be specified as well as possible in the framework of a system and the detailed design has to be checked separately to fulfil their specifications in the "bigger picture".

The choice of an adequate design modelling tool is not sufficient to design a faultless system. It is equally important to employ a proper design process. Indeed, faults can also be due to the designers' inability of deducing a correct model due to the expressive means available or due to an inappropriate choice of design phases. One of the prominent design processes for faultless systems is the standard SIL-based process model presented in the next section.

Example: Coding Rules for Safety-critical Applications

Most serious software development projects use coding guidelines. In connection with a consistent design method and process, they are very important to ensure coherent software artifacts in a larger and/or evolving project team. They define the ground rules for the software to be written "by hand": how it should be structured, which language features should be used and how. According to [26] there is, surprisingly, little consensus on what a good coding standard is. In this article, ten rules were defined with special emphasis on safety-critical applications:

1. Restrict all code to very simple control flow constructs — do not use *goto* statements, *setjmp* or *longjmp* constructs, or direct or indirect recursion.
2. Give all loops a fixed upper bound. It must be at least trivially possible to statically prove that a loop cannot exceed a pre-set upper bound for the number of iterations.
3. Do not use dynamic memory allocation after initialisation.
4. No function should be longer than what can be printed on a single sheet of paper in a standard format with one line per statement and one line per declaration (typically this means a maximum of 60 lines of code per function).
5. The code's assertion density should average to a minimum of two assertions per function. Assertions must be used to check for anomalous conditions that should never happen in real-life executions. They must be free of side effects and should be defined as Boolean tests. When an assertion fails, an explicit recovery action must be taken, such as returning an error condition to the caller of the function that executes the failing assertion. Any assertion for which a static checking tool can prove that it can never fail or never hold violates this rule.
6. Declare all data objects at the smallest possible level of scope.
7. Each calling function must check the return value of non-*void* functions, and each called function must check the validity of all parameters provided by the caller.
8. The use of a preprocessor must be limited to the inclusion of header files and simple macro definitions. Token pasting, variable argument lists (ellipses) and recursive macro calls are not allowed. All macros must expand into complete syntactic units. The use of conditional compilation directives should be kept to a minimum.

9. The use of pointers must be restricted. Specifically, no more than one level of de-referencing should be used. Pointer de-reference operations may not be hidden in macro definitions or inside *typedef* declarations. Function pointers are not permitted.
10. All code must be compiled from the first day of development, with all compiler warnings enabled at the most pedantic setting available. All code must compile without warnings. All code must also be checked daily with at least one, but preferably more than one, strong static source code analyser and should pass all analyses with zero warnings.

Of course the choice of a programming language is a key consideration in this concern. Since for programming aerospace, industrial automation and automotive applications mostly C is used, in this case it was also the language chosen and the rules apply to it. The benefit of using a smaller number of coding rules lies in the fact that it is more likely that they are going to be used as opposed to hundreds of rules and guidelines. The author of [26] claims, that the quality of source code improved within the NASA/UPL Laboratory for Reliable Software by using these rules.

While the first few rules ensure the creation of clear and transparent control flow structures that are easier to build, test and analyse, the other rules address some "fancy" features of the language C, which work in principle, but are quite error-prone. Dynamic memory allocation is discouraged because of the run-time problems with temporally deterministic memory management in environments with potentially very limited memory resources. Some rules address the fault detection and removal techniques described later. These rules correlate very strongly with the guidelines and regulations that have been assembled and classified in the standard IEC 61508 described previously.

Fault Removal

The fault prevention techniques presented so far are useful during specification and design. Here we examine a designed system (model) in order to detect the possible presence of residual faults. Some techniques require the presence of specifications, while others are derived from the designed system. Their demonstration strength ranges from *partial functional simulation* to *complete formal proof.* They may apply to a system that has not yet been fully designed or implemented (in hardware and software) or to a final product.

Verification without specifications aims at checking the design model (system) for the existence of undesired properties such as deadlocks or non-determinism. Verification with specifications is possible by the following methods:

- Bottom-up reverse transformation of the designed system/model into a specification and its comparison with the actual specification
- Top-down verification by bivalent specification to designed system/models and their comparison by simulation sequences or by checking functional static and/or dynamic properties

A simulation sequence, also known as "test sequence", generally consists of tuples of inputs and expected outputs, which are applied to the design model (system) and compared with the results obtained. If they are equal to the expected outputs, the system "conforms to the specifications". For real-time systems, they should be enhanced by the time dimension. The level of trust in the results depends on the quality of the test sequence. Generally, it depends on the level of structural knowledge about the design model (system) — we speak about black-box (no knowledge), grey-box (little knowledge) and white-box (complete knowledge) tests.

Short Test Sequences

A system may be designed in many different ways depending on the nature of the problem domain, required functionality, and a number of QoS parameters resulting in very different implementations: circuits containing electronic and mechanical components, logic gates (arrays), or sequential program code and very likely combinations thereof. Specific design methods naturally integrate the previously stated remarks on testability connecting the most suitable test methods with the respective design methods.

Often ideas on testability can be transferred from mechanical to electrical engineering and further to program design and test, which makes integration tests easier with hybrid systems being composed of all three and possibly additional (e.g. chemical) kinds of devices.

The subsequent examples show how fault detection and removal techniques have been integrated with specific design methods. This, however, does not mean that the use of the principles presented is restricted to the domains indicated.

Built-In Test (BIT): This technique consists of adding a specific standard interface to the system under test, which controls and facilitates access from the external tester, thus increasing controllability and observability (see Fig. 6.2). As a consequence, the test sequences are simpler, and their application is facilitated. Hence, testing is also simpler. The drawback is that the technique introduces some additional overhead into hardware components and/or software code.

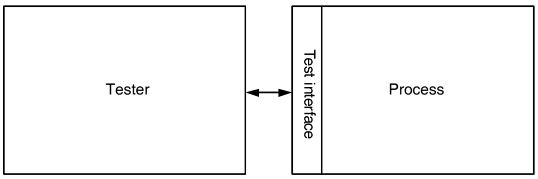

Fig. 6.2. BIT test schema

Boundary Scan: In IEEE 1194-1 the IEEE Joint Test Action Group (JTAG) defined a standardised interface to test integrated circuits. Today, the majority of integrated circuit manufacturers uses this standard to test their designs of application-specific integrated circuits, microprocessors and microcontrollers.

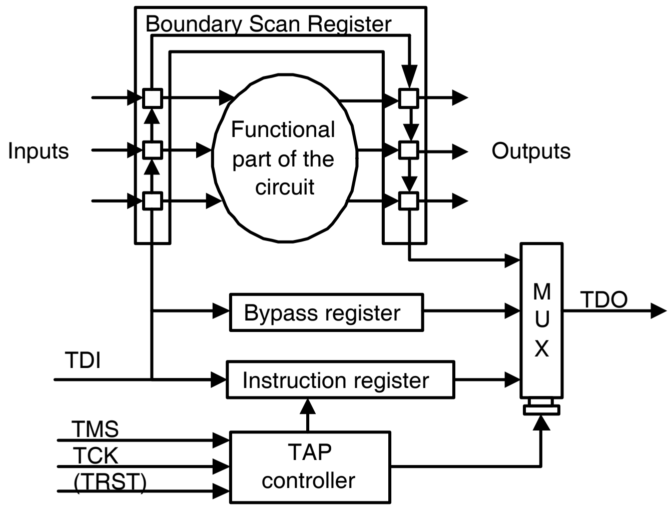

Fig. 6.3. Schema of Boundary Scan Register

addition of some test inputs/outputs and an internal logic:

Test bus ("Test Access Port" (TAP)) composed of a series of input/output signals (TDI, TDO) and a test clock (TCK)

Integrated logic module comprising:

1. A series register ("Boundary Scan Register") to read in the test inputs and read out the test results
2. A bypass register to reach other modules located below in the test chain of the system's modules
3. An automaton (TAP Controller) associated with an instruction register, which may process certain test operations. It controls the various operations in normal or test mode. In normal mode, the boundary scan register cells are set so that they do not affect the circuit. In test mode, however, the cells are latched in a way enabling a data stream between them — once a complete data word has been read in, it can be latched into place. The cell at the destination pin of the circuit can be programmed to verify if the circuit trace correctly interconnects the two pins (input and output).

Each cell (see Fig. 6.4) of the *Boundary Scan Register* has four modes of functioning. In "normal mode" (1) the cell's output multiplexer (*Output MUX*) transfers the data coming from an input pin of the circuit to the output of the cell (*Data Out*), which is connected to the input of the core logic; in "update mode" (2) the output multiplexer sends the contents of the *Parallel Output Register* to the cell's output; in "capture mode" (3) the input data (*Data In*) is directed by the input multiplexer (*Input MUX*) to the *Shift Register* in order to be loaded when the *Clock DR* occurs; in “shift mode” (4) the bit of each cell of the *Boundary Scan Register* (its *Shift Register*) on the *Inputs* side is sent to the following cell via the *Scan Out* line, whilst the signal *Scan In* coming from the previous cell is loaded into the flip-flop of the cell’s *Shift Register*. The output cells use the same principle. As the cells can be used to force data into the circuit, they can set test conditions. The relevant states can then be fed back into the test system by clocking the data word back so that it can be analysed.

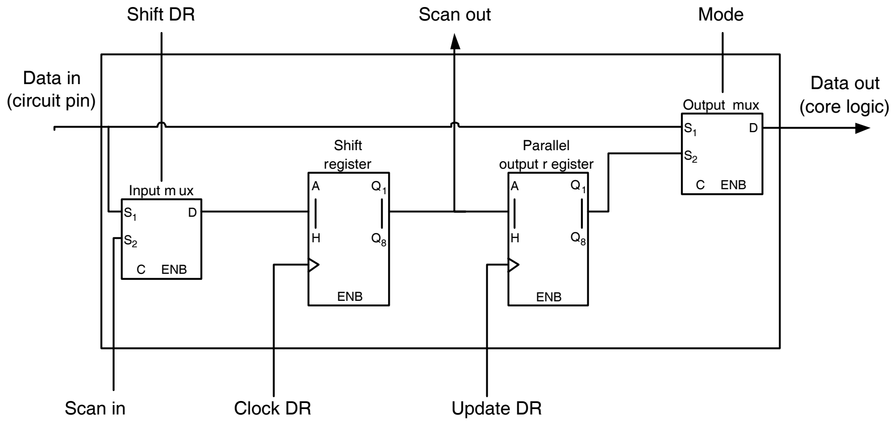

Fig. 6.4. Boundary Input Cell

The circuit logic described allows one to control all inputs and outputs of the circuit tested with the help of a shift register, which can be loaded and unloaded in series. It is also possible to pass information across a circuit in order to reach a circuit situated below it, or to receive information coming from this circuit. Adopting this technique, it is possible for a test system to gain test access to a circuit board. As most current boards are very densely packed with components and lines, it can be very difficult for test systems to access the relevant areas of a board via probes for test purposes. The boundary scan makes this possible. In some cases the boundary scan technique is not applied to a circuit as a whole. Then, we say that this circuit implements a “partial scan” as opposed to a “full scan”.

The same hardware circuit test logic can also be applied to software. “Scan design” is appropriate for circuits or software modules having a few inputs and outputs, since otherwise too long test sequences would result. A natural approach to solve this problem is to partition circuits/software into parts called “scan domains”. Once the partial “unit tests” are complete, “integration tests” can be performed.

Built-In Self Test (BIST): The BIT technique (see Fig. 6.2) requires an external tester (e.g. “off-chip test” in hardware design), while with the BIST technique (see Fig. 6.5) the tester is integrated into the tested system (e.g. “on-chip test” in hardware design). The method requires a greater investment in design. The BIST approach improves the principle of integrating the tester functions into the system under test. This solution is only acceptable if its complexity and the expenditure are not excessive. It is justified when the coverage of faults tested is increased considerably. The technique allows the system to test itself, usually off-line, with the normal function of the system being suspended during the test. Usually, the test operations are run during power-up or maintenance operations. Some fault tolerance techniques require on-line monitoring of inputs and outputs. In this case, BIST functions are integrated into the normal functionality of the system under test. BIST techniques are integrated in more and more industrial products.

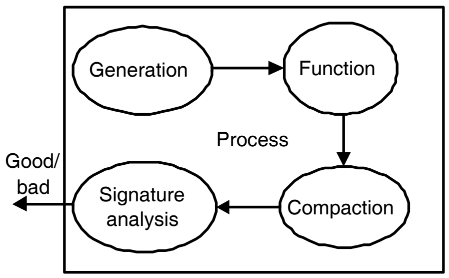

Fig. 6.5. The BIST test schema

As an example, the generation of pseudo-random test patterns for BIST employing a Linear Feedback Shift Register (LFSR) is presented. The 3-bit LFSR shown in Fig. 6.6 is a synchronous sequential circuit, using D flip-flops and XOR gates, which generates a pseudo-random output pattern of 0s and 1s. Suppose the circuit is initialised in state (Q1,Q2,Q3)=(1,1,1); it produces a cyclic output sequence with an (Q1,Q2,Q3) output vector for any clock pulse. Such an LFSR can be used as a test sequence generator in an integrated circuit (DUT – Device Under Test) as shown in Fig. 6.7. It can also be used as a compaction circuit (PSA – Parallel Signal Analyser) in order to reduce the length of the output sequence (see Fig. 6.8).

Fault Tolerance

Fail-safe or *fail-secure* is the property of a system/device that, if (or when) it fails, it fails in such a way not to cause harm, or only minimum harm, to other devices or danger to personnel.

Fail-safe operation means automatic protection of programs and/or processing systems when a hardware or software failure is detected in a computer system.

Fault tolerance is the property that enables a system to continue operating properly upon occurrence of a failure in (or of one or more faults within) some of its components.

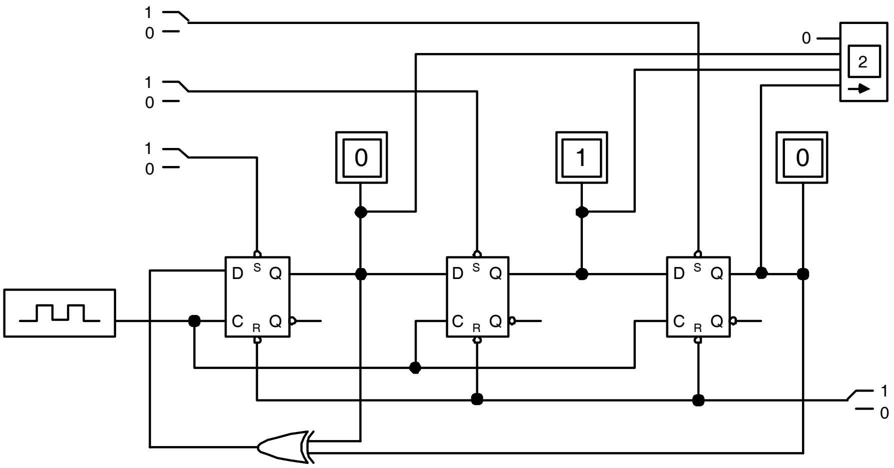

Fig. 6.6. 3-bit LFSR

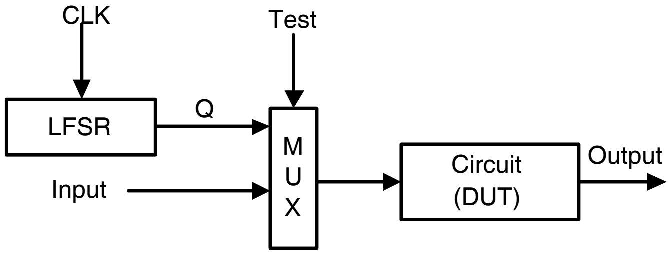

Fig. 6.7. Test sequence generation

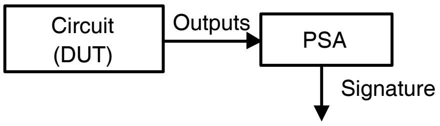

Fig. 6.8. Test result compaction

Protective *fault tolerance mechanisms* are defined and implemented during the system creation phases of the life-cycle, whereas their actions are effective during the operational phase. Fault prevention and removal techniques allow one to increase reliability (reducing the probability of fault occurrences), or availability in the case of repairable systems (e.g. detect-and-repair mechanisms). Observing safety criteria leads one to examine techniques related to *on-line testing* and *fail-safe design*. Here, we want to provide a system designed with the highest degree of dependability by integrating mechanisms that allow for full continuation of its mission despite the presence of faults, thus increasing its integrity. Such mechanisms may deliver full service at no reduction in performance. Due to the presence of faults, however, the performance usually decreases to an acceptable level ("graceful degradation"). Employing fault tolerance techniques has a direct positive impact on *safety*, *reliability*, and *availability*.

In the sequel, the basic characteristics of fault tolerance are dealt with in more detail. Fault-tolerant systems are also characterised in terms of planned and unplanned service outages. They are usually regarded/measured from the application point of view, and quantified by *availability* as a percentage (e.g. an availability of 95.9% means that a system is up this fraction of time).

Fault tolerance by replication: The first fundamental characteristic of fault tolerance is "no single point of failure". It is addressed by spare components (replicas) in three ways:

1. *Replication:* providing multiple identical instances of the same system or device, directing tasks or requests to all of them in parallel and choosing the correct result on the basis of a quorum
2. *Redundancy:* providing multiple identical instances of the same system or device and switching to a remaining one in case of a failure (failover)
3. *Diversity:* providing multiple different implementations of the same specification, and using them like replicated systems to cope with errors in a specific implementation

A good example of replication is a redundant array of independent disks (RAID) representing a fault-tolerant storage device using data redundancy.

A *lockstep fault-tolerant system* consists of replicated elements operating in parallel. At any time, all replicas of each element should be in the same state. The same inputs are provided to each replica, and the same outputs are expected. The replicas' outputs are compared using a voting circuit. A system with two replicas for each element is termed *Dual Modular Redundant* (DMR). In this case the voting circuit can detect a mismatch, only, and recovery relies on other methods. A system with three replicas for each element is termed *Triple Modular Redundant* (TMR). The voting circuit can then determine which replication is in error and output the correct result based on a two-to-one vote. After that, the voting circuit usually switches to DMR mode. This model can be applied to a larger number of replicas, e.g. *pair-and-spare*. Here, two replicated elements operate in lockstep as a pair, with a voting circuit that detects any mismatch between their operations and outputs a signal indicating when there was an error. Another pair operates exactly in the same way. A final circuit selects the output of the pair not being in error.

No single point of repair: With this option enabled, a system must continue to operate without interruption during the repair process if it experiences a failure.

Fault isolation to the failing component: When a failure occurs, a system must be able to isolate the failure to the erroneous component. This requires dedicated failure detection mechanisms to be added, solely for the purpose of fault isolation.

Fault containment: This is to prevent the propagation of failures. With some failure handling mechanisms a failure may be propagated to the rest of a system, possibly causing it to fail completely. Mechanisms that isolate a failing component to protect the system are required.

Reversion modes: With some failure mechanisms the survivability of a system, its operator, or the final results may be endangered. To prevent such losses, mission-critical systems (e.g. in defence or avionics) provide for safe modes using *interlock* mechanisms or software.

Example: Safety Shell

Since the two most important properties of real-time systems are dependability and timeliness, for their safety-oriented design a so-called system "safety shell" has been defined [45, 46]. It combines a suitable combination of the above listed options and mechanisms (replication, no single point of failure, fault containment) to ensure timely, fault-tolerant execution.

The software safety shell shown in Fig. 6.9 depends on a guard to ensure safe execution in a system. The guard is a low-level construct that detects faults and, then, forces the system to transfer to a safe state instead of a hazardous one. As can be seen, the "Primary Control" is guarded from erroneous input from the environment by the "State Guard". Its *functional correctness* is ensured by defining well formed algorithms with strict limitations on their I/O range, and "Exception Handlers" maintaining safe and stable states of operation. At the same time, the timely, synchronous operation of the system is guarded by the "Timing Guard", implemented by, e.g. "watchdogs" for critical operations.

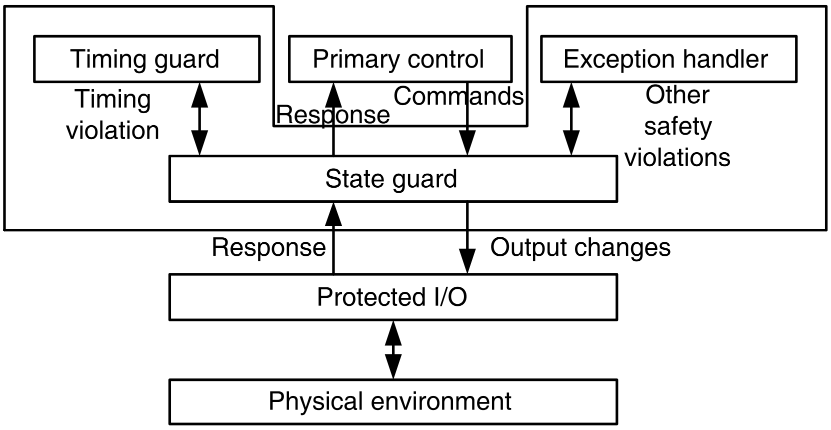

Fig. 6.9. Safety shell

Fault Forecasting

Fault forecasting aims to estimate the presence of faults (number and severity). It implies the means that allow the evaluation or measurement of *reliance*. The assessment approaches available can be categorised into quantitative and qualitative dependability assessment.

The objective of *qualitative assessment* is to examine faults, errors, potentiality of failures, and their effects. The assessment methods are twofold — deductive and inductive. In the deductive approach, the failures anticipated are derived from faults and errors. In the inductive approach, the potential failures are examined from present faults or errors.

In the standard EN 13306 [20] *reliability* is defined as: *"The ability of an item to perform a required function under given conditions for a given time interval,"* where item denotes a device (product) considered for maintenance or maintenance management. *Quantitative assessment* considers reliability as a function of time which expresses the conditional probability that a system has survived in a specified environment until time *t*, given that it was operational at time *0*. This definition is the origin of the quantitative QoS parameters (e.g. MTBF, MTTF and MTTR).

A product's reliability function does not increase with time. Its decreasing tendency is due to the subjacent phenomenon of degradation of electronic devices. In the case of software, once this function has reached a certain level, it remains constant due to the absence of ageing (not considering maintenance operations).

Different types of *reliability tests* are carried out to perform *reliability evaluation* with respect to test stop conditions such as fixed duration, fixed number of faults reached, result-based end criteria, and combinations thereof, and with the possibility of accelerated — *avalanche/stress* — tests.

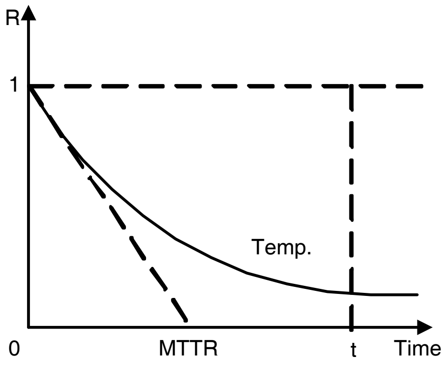

Fig. 6.10. The exponential law

Reliability is analysed by *reliability models*, which are mathematical functions of time. The *exponential law* is the simplest of these laws. It provides the probability of survival by an exponential function, which decreases with time (see Fig. 6.10). The *failure rate* λ expresses the probability of failures occurring per hour, and is generally considered to be a constant throughout time. The $R(t)$ law is often associated with the simple estimators:

MTTF "Mean Time To Failure" (also called Mean Time to First Failure – MTFF) for non-repairable systems (e.g. a mission terminating as soon as a breakdown happens)

MTBF "Mean Time Between Failures" for repairable systems (e.g. a component, which broke down, is repaired/replaced and put back into operation)

If a reliability function described by the exponential law has a constant failure rate, the average value of this function is: MTBF (or MTFF) $= \frac{1}{\lambda}$ and is expressed in powers of 10 hours.

Failure rates are generally estimated from survival tests applied to significant large samples of components. The duration of these tests is short as compared to a product's normal life-cycle. Then, by using acceleration factors, the failure rates of, e.g. electrical circuits are deduced. These tests are thus called *accelerated tests*. A component's degradation process can be accelerated by increasing the temperature, and also by increasing the voltage of the power supply. Most failure mechanisms in integrated circuits, for instance, are based on physiochemical reactions that can be accelerated (*stress test*) by temperature in accordance with the Arrhenius equation: $\lambda_b = A \cdot e^{-\frac{E}{kT}}$, where λ_b is the process failure rate, E is the activation energy (in $[eV]$), k is the Boltzmann constant ($8.6 \cdot 10^{-5}$ $[V/K]$), T is the temperature (in $[K]$) and A is a constant. The real value of a given circuit's λ is deduced from $\prod \lambda_{b_i}$) where the b_i identify the components of the circuit.

6.2 Security-oriented Design

Addressing security generally comprises the continuous cycle of risk management and response planning. Because of changes in the environment or the system itself, which may also be time-related, it is impossible to set a fixed strategy. Hence, a cyclic approach to security management [3] has proven most successful. However, a number of guidelines and standards has been developed for security-aware design.

Risk management and the response cycle comprise the following steps: (1) upon detection, a threat agent gives rise to a threat; (2) based on the threat, vulnerability is explored; (3) a discovered vulnerability leads to a risk, which could damage assets or cause exposure, depending on the level/kind of the risk; and (4) the threat must be safeguarded by the threat agent. There are two ways a threat agent can respond to a threat: (1) by mitigation to minimise loss or probability of exposure, or (2) by acceptance with active or passive handling of the security violation. In the latter case, depending on the type of system and the kind of security violation, proper handling is chosen. With intrusions via the Internet active handling means, for instance, to give an intruder bogus results in order to sustain security and discover its identity, whereas passive handling means ending the intrusion process in a controlled way in order to prevent it from causing any (further) damage to the security of the system.

Contingency planning has been developed to systematically integrate security management into technical and business processes. It is composed of four major steps:

1. *(System/Business) Impact Analysis (BIA):* consisting of threat attack identification and prioritisation, business unit analysis, attack success scenario development, potential damage assessment and subordinate plan classification
2. *Incident Response Planning:* comprising incident planning, incident detection, incident reaction and incident recovery
3. *Disaster Recovery Planning:* creating plans for disaster recovery, crisis management and recovery operations
4. *(Business) Continuity Planning:* establishing continuity strategies, plans for continuity of operation and continuity management

The areas of security pertaining to real-time systems can be grouped into the following two main categories:

1. *Environmental:* physical security, personal security, operation security and communication security
2. *Technical:* network security, information security and computer security

Measures coping with environmental threats to security are taken in order to prevent damage to a system either caused by natural disasters and atmospherical effects or the logistics behind the operation or by the personnel. Although very important, these issues are not discussed here — they represent the responsibility of management and security engineers. The issues to be considered here represent the technical aspects of ensuring security of operation.

Technical security issues with computing systems encompass mainly three areas:

1. Computer system security
2. Computer network security
3. Information security

As computer system security the security of a computing system's operation is considered, by taking failure probabilities of, e.g. processors and data storages into account. These issues are handled predominantly by choosing an appropriate architecture for the computing system to allow for its proper operation, even in the case of component failure, by component replication as well as by appropriate failure diagnostics and handling. Here it is of utmost importance not to allow a single point of failure in security-critical designs. Usually the computing system's operation shall be degraded gracefully until a malfunctioning component is repaired or replaced; however, it shall not deny service. Here, real-time restrictions apply to worst-case response times and worst-case repair times.

Computer network security is partly included in computer system security, especially with distributed computing systems. The issues here are mainly the same, except that they involve hardware and software interfaces and drivers, which must ensure data integrity and communication network availability. The basic principle of replication to avoid a single point of failure also pertains to communication networks.

It is only achieved by different means, viz., line replication. Communication lines are usually chosen for transmission speed during normal operation, but for safety during the reduced (on-failure) mode of operation. An issue partly also belonging to information security is here dealt with as well, namely, data security. The information transferred on a network is not only encoded in packages to be correctly processed by the communication drivers, but also encrypted in order to minimise the possibility of extracting information from the data packages by intruders. Real-time restrictions applying here concern network propagation delays and times for re-routing messages on "healthy lines" in the case of broken connection lines.

Information security comprises information integrity as well as confidentiality. Integrity is ensured by the former two areas of technical security, whereas confidentiality is achieved by means of authentication and authorisation. These mechanisms are meant to ensure that only authorised individuals have access to sensitive information like, e.g. records of production processes, medical examinations, or financial transactions. To authorise them, these individuals or organisations need to be authenticated by the data host. Their identity is determined by lock-key mechanisms, which must be safeguarded in order not to reveal the identities to third parties who might misuse the information. The identification mechanisms involve biometrical information (e.g. fingerprints or iris scans) or electronic keys. After an individual is authenticated in a computing system the data flow transferring information across the network is encrypted. The time for authentication needs to be short to prevent brute force attacks. Also the durations of activities need to be monitored, e.g. in order to prevent that someone's data session is taken over by an intruder.

6.2.1 Security Layers

Since security attacks have become more frequent and more sophisticated, investing in single-layer security systems is of limited use. Security experts have built a layered security architecture [2] (see Fig. 6.11) and associated different security management strategies according to their respective strengths and limitations with each individual layer:

Security Policy Layer: this should encompass all aspects of employee awareness of security and responsibility to network, e-mail, Internet and password usage.

Physical Layer: this keeps computers (servers, workstations, portable devices, software media, data back-up media, switches, routers, firewalls, etc.) locked down and safe from physical theft and intrusion. Any device must have a designated owner who is responsible for its security. The owner should have appropriate resources, skills and information to fulfil this responsibility. Network equipment and software should be restricted to authorised personnel.

Data Layer: this is limited to the accessibility of data on any given network. The desired outcome is one that restricts data access to only those users who are required to have access. This protects privacy and provides for accountability. Web application gateways, e-mail spam filters, XML security systems and Secure Sockets Layer (SSL) virtual private networks help to ensure that application traffic is clean, efficient and secure.

Application Layer: this provides an automated security layer to protect configurations on hosts and includes host-based anti-virus applications, intrusion prevention software, spyware tools and personal firewalls. With the most advanced set of tools, these products provide essential "last-resort" security for applications and networks to thwart any potential threat. As with most software, however, human intervention is required to ensure that the solution is constantly updated.

Network Perimeter Layer (NPL): this utilises hardware and conceptual design to provide a layer of protection from outside a network. To create this layer of protection, the NPL utilises firewalls, Virtual Private Networks (VPNs), routers, intrusion detection and prevention software and web content filtering.

Management Layer: this is critical to the continued security of an overall network

- to consolidate the approach to security management, assess the overall vulnerability, and manage patches and updates carefully;
- to persistently monitor all security layers for compliance and vulnerabilities. This includes creating a security framework that makes it possible to identify potential threats early, accurately analyse risks from emerging threats and swiftly develop effective remedial strategies as well as protect an entire organisation, from the borders of the corporate network down to each individual computing system (component). In general, this requires supervision to ensure consistency in assessing overall vulnerability and managing patches and updates for any software and policy.

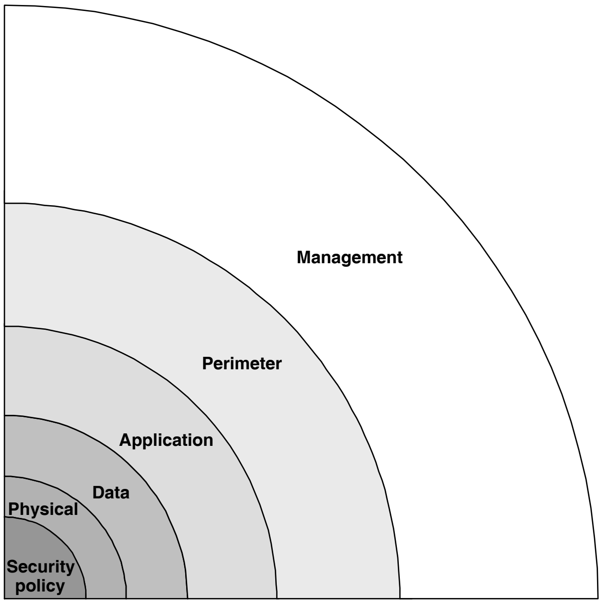

Fig. 6.11. Security layers

From the security point of view, real-time systems do not differ from other computing systems. Sustaining their QoS through security measures is the primary goal, and the tools and methods to achieve this are basically the same. Hence, there is no special notion of real time here, and no distinction is made with respect to the concepts of security. Some security mechanisms do have real-time constraints, but they do not pertain to the systems or the applications. One may establish trade-offs between levels of security and QoS levels, however, since security mechanisms also require computing time and resources, system performance may be affected. In the sequel the most important technical security mechanisms are assessed in more detail.

Encryption: Typical examples of encryption mechanisms for networked systems are Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL). They represent cryptographic protocols providing secure communications on the Internet (for web browsing, e-mail and other data transfers).
The TLS protocol allows applications to communicate across a network in a way that prevents eavesdropping, tampering and message forgery. It provides endpoint authentication and communication privacy over the Internet using cryptography. Typically, only the server is authenticated (i.e. its identity is ensured), while the client remains unauthenticated. The next level of security – in which both conversation partners are sure with whom they are communicating – is known as *mutual authentication*. It requires Public Key Infrastructure (PKI) deployment to clients unless Pre-Shared Key cypher suits (TLS-PSK) or Secure Remote Password (TLS-SRP) are used, which provide strong mutual authentication without needing to deploy a PKI. The TLS protocol involves three basic phases:

1. Peer negotiation for algorithm support
2. Public key exchange and certificate-based authentication
3. Symmetric cipher encryption

During the first phase, the client and server negotiate cipher suites, which combine one cipher from each of the following:

- Public-key cryptography (e.g. RSA, Diffie-Hellman, DSA)
- Cryptographic hash function (e.g. MD2, MD4, MD5, SHA)
- Symmetric ciphers (e.g. RC2, RC4, IDEA, (Triple) DES, AES, Camellia)

Usually, attacks against a “cryptosystem” without known vulnerability, e.g. Digital Encryption Standard (DES) [64] or Advanced Encryption Standard (AES) [51], are brute force ones trying to find secret keys by exhaustive search in the key spaces. Real-time restrictions apply in order to prevent attackers from being given enough time to “crack” the security keys.

Access Control: Due to the large variety of access control mechanisms from password protection to access control devices (e.g. fingerprint readers, iris scanners, RFID), depending on the level of required security this term does not uniquely identify the access control process. In any case, successful access control encompasses both authentication and authorisation.
The different types of *authentication* mechanisms relate to something you know, something you have, something you are, or something you produce. They have

time limitations, as otherwise we would not be able to distinguish authentication requests in the course of a lengthy session, during which an attacker could run a series of access attempts and eventually succeed.

Authorisation must provide services to any authenticated user, members of a group, and/or authorisation across multiple systems. Dial-up protection is used to prevent intrusion by unauthorised individuals, which may use a network's Internet connection to gather information or cause damage. Timing limitations also pertain to authorisation. During a certain period of time, an individual shall receive only one authorisation. Also, during a session, the authorisation system may perform an "inactivity logout" canceling the authorisation after a lengthy period of inactivity. In practice this could otherwise mean an open gate for an intruder accessing the individual's computer during that time.

Since embedded networks also have some Internet connectivity for monitoring or maintenance, these connections are targeted by attackers to collect data from the embedded network or computer systems. In order to detect attackers, scanning and analysis tools may be employed to sense extraordinary executions or information leaks. Intrusion detection systems build on sophisticated authentication and access control algorithms as well as on encryption systems to prevent sensible data leaving intranets, which could be intercepted on the Internet.

Firewalls: In information security, a firewall is any device that prevents a specific type of information from moving between two networks. A firewall may be a separate computer system, a service running on an existing router or server or a separate network containing a number of supporting devices. Access restrictions are applied using authentication mechanisms.

6.3 Concluding Remarks on Design for QoS

All aspects of system design and development discussed are relevant for a designed product's QoS. Although they address different QoS criteria, they all contribute to the overall QoS. A popular fault prevention technique is to introduce *capacity reserves* into the consideration of physical and timing constraints, which reduce the risk of failure. Apart from introducing resource redundancy, the mentioned temporal analysis tools may be helpful to estimate appropriate slack times in the prescribed timing constraints rather than just using rules of thumb based on expert knowledge. In general, however, fault prevention must be viewed in a broader context and dealt with in multiple phases of system development. The concepts of BIT and BIST originate from hardware diagnostics, but their principles can be transferred to software provided that the number of inputs and outputs is limited. In safety-critical environments, the considerations for Safety Integrity Level, life-cycle and fault tolerance mechanisms determine the way systems are devised and maintained. In security-critical applications, the mentioned security measures need to be employed to prevent damage to human health and machinery, or undesired data leakage of sensible information.