Chapter 1
The Real-Time Environment

Overview The purpose of this introductory chapter is to describe the environment of real-time computer systems from a number of different perspectives. A solid understanding of the technical and economic factors that characterize a real-time application helps to interpret the demands that the system designer must cope with. The chapter starts with the definition of a real-time system and with a discussion of its functional and meta-functional requirements. Particular emphasis is placed on the temporal requirements that are derived from the well-understood properties of control applications. The objective of a control algorithm is to drive a process such that a performance criterion is satisfied. Random disturbances occurring in the environment degrade system performance and must be taken into account by the control algorithm. Any additional uncertainty that is introduced into the control loop by the control system itself, e.g., a non-predictable jitter of the control loop, results in a degradation of the quality of control.

In the Sects. 1.2 to 1.5 real-time applications are classified from a number of viewpoints. Special emphasis is placed on the fundamental differences between *hard* and *soft* real-time systems. Because soft real-time systems do not have severe failure modes, a less rigorous approach to their design is often followed. Sometimes resource-inadequate solutions that will not handle the rarely occurring peak-load scenarios are accepted on economic arguments. In a hard real-time application, such an approach is unacceptable because the safety of a design in all specified situations, even if they occur only very rarely, must be demonstrated *vis-a-vis* a certification agency. In Sect. 1.6, a brief analysis of the real-time system market is carried out with emphasis on the field of embedded real-time systems. An embedded real-time system is a part of a self-contained product, e.g., a television set or an automobile. Embedded real-time systems, also called *cyber-physical* (CPS) systems, form the most important market segment for real-time technology and the computer industry in general.

H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Real-Time Systems Series, DOI 10.1007/978-1-4419-8237-7_1,

© Springer Science+Business Media, LLC 2011

1.1 When Is a Computer System Real-Time?

A *real-time computer system* is a computer system where the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical time when these results are produced. By *system behavior* we mean the sequence of *outputs in time* of a system.

We model the flow of time by a directed time line that extends from the past into the future. A cut of the time line is called an *instant*. Any *ideal* occurrence that happens at an *instant* is called an *event*. Information that describes an event (see also Sect. 5.2.4 on event observation) is called *event information*. The present point in time, *now,* is a very special event that separates the *past* from the *future* (the presented model of time is based on Newtonian physics and disregards relativistic effects). An *interval* on the time line, called a *duration*, is defined by two events, the *start event* and the *terminating event* of the interval. A digital clock partitions the time line into a sequence of equally spaced durations, called the *granules* of the clock, which are delimited by special periodic events, the *ticks* of the clock.

A real-time computer system is always part of a larger system – this larger system is called a *real-time system* or a *cyber-physical system*. A real-time system changes as a function of physical time, e.g., a chemical reaction continues to change its state even after its controlling computer system has stopped. It is reasonable to decompose a real-time system into a set of self-contained subsystems called *clusters*. Examples of clusters are (Fig. 1.1): the physical plant or machine that is to be controlled (the *controlled cluster*), the real-time computer system (the *computational cluster*;) and, the *human operator* (the *operator cluster*). We refer to the *controlled cluster* and the *operator cluster* collectively as the *environment* of the *computational cluster* (the real-time computer system).

If the real-time computer system is *distributed* (and most of them are), it consists of a set of (computer) *nodes* interconnected by a real-time communication network.

The interface between the human operator and the real-time computer system is called the *man–machine interface,* and the interface between the controlled object and the real-time computer system is called the *instrumentation interface*. The man–machine interface consists of input devices (e.g., keyboard) and output devices (e.g., display) that interface to the human operator. The instrumentation

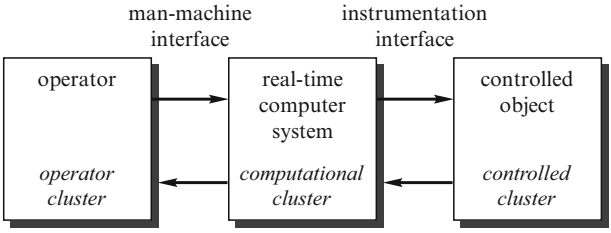

Fig. 1.1 Real-time system

interface consists of the sensors and actuators that transform the physical signals (e.g., voltages, currents) in the controlled cluster into a digital form and *vice versa*.

A real-time computer system must react to stimuli from its environment (the controlled cluster or the operator cluster) within time intervals dictated by its environment. The instant when a result must be produced is called a *deadline*. If a result has utility even after the deadline has passed, the deadline is classified as *soft*, otherwise it is *firm*. If *severe consequences* could result if a firm deadline is missed, the deadline is called *hard*.

> **Example:** Consider a traffic signal at a road before a railway crossing. If the traffic signal does not change to *red* before the train arrives, an accident could result.

A real-time computer system that must meet at least one hard deadline is called a *hard real-time computer system* or a *safety-critical real-time computer system*. If no hard deadline exists, then the system is called a *soft real-time computer system*.

The design of a hard real-time system is fundamentally different from the design of a soft real-time system. While a hard real-time computer system must sustain a guaranteed temporal behavior under all specified load and fault conditions, it is permissible for a soft real-time computer system to miss a deadline occasionally. The differences between soft and hard real-time systems will be discussed in detail in the following sections. The focus of this book is on the design of hard real-time systems.

1.2 Functional Requirements

The functional requirements of real-time systems are concerned with the functions that a real-time computer system must perform. They are grouped into data collection requirements, direct digital control requirements, and man–machine interaction requirements.

1.2.1 Data Collection

A controlled object, e.g., a car or an industrial plant, changes its state as a function of time (whenever we use the word *time* without a qualifier, we mean *physical time* as described in Sect. 3.1). If we freeze the time, we can describe the current state of the controlled object by recording the values of its state variables at that moment. Possible state variables of a controlled object *car* are the position of the car, the speed of the car, the position of switches on the dashboard, and the position of a piston in a cylinder. We are normally not interested in *all* state variables, but only in the *subset* of state variables that is *significant* for our purpose. A significant state variable is called a *real-time (RT) entity*.

Every RT entity is in the *sphere of control (SOC)* of a subsystem, i.e., it belongs to a subsystem that has the authority to change the value of this RT entity (see also Sect. 5.1.1). Outside its sphere of control, the value of an RT entity can be observed, but its *semantic content* (see Sect. 2.2.4) cannot be modified. For example, the current position of a piston in a cylinder of the engine is in the sphere of control of the engine. Outside the car engine the current position of the piston can only be observed, but we are not allowed to modify the *semantic content* of this observation (the representation of the *semantic content* can be changed!).

The first functional requirement of a real-time computer system is the observation of the RT entities in a controlled cluster and the collection of these observations. An observation of an RT entity is represented by a *real-time (RT) image* in the computer system. Since the state of a *controlled object* in the *controlled cluster* is a function of real time, a given RT image is only *temporally accurate* for a limited time interval. The length of this time interval depends on the dynamics of the controlled object. If the state of the controlled object changes very quickly, the corresponding RT image has a very short *accuracy interval*.

> **Example:** Consider the example of Fig. 1.2, where a car enters an intersection controlled by a traffic light. How long is the observation *the traffic light is green* temporally accurate? If the information *the traffic light is green* is used outside its accuracy interval, i.e., a car enters the intersection after the traffic light has switched to *red*, an accident may occur. In this example, an upper bound for the accuracy interval is given by the duration of the yellow phase of the traffic light.

The set of all temporally accurate real-time images of the controlled cluster is called the *real-time database*. The real-time database must be updated whenever an RT entity changes its value. These updates can be performed periodically, triggered by the progression of the real-time clock by a fixed period (*time-triggered (TT) observation*), or immediately after a change of state, which constitutes an event, occurs in the RT entity (*event-triggered (ET) observation*). A more detailed analysis of time-triggered and event-triggered observations will be presented in Chaps. 4 and 5.

Signal Conditioning. A physical sensor, e.g., a thermocouple, produces a *raw data* element (e.g., a voltage). Often, a sequence of raw data elements is collected and an averaging algorithm is applied to reduce the measurement error. In the next step the raw data must be calibrated and transformed to standard measurement units.

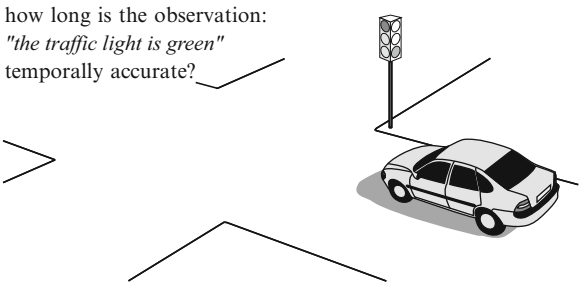

Fig. 1.2 Temporal accuracy of the traffic light information

The term *signal conditioning* is used to refer to all the processing steps that are necessary to obtain meaningful *measured data* of an RT entity from the raw sensor data. After signal conditioning, the measured data must be checked for plausibility and related to other measured data to detect a possible fault of the sensor. A data element that is judged to be a correct RT image of the corresponding RT entity is called an *agreed data element*.

Alarm Monitoring. An important function of a real-time computer system is the continuous monitoring of the RT entities to detect abnormal process behaviors.

> **Example:** The rupture of a pipe, a *primary event*, in a chemical plant will cause many RT entities (diverse pressures, temperatures, liquid levels) to deviate from their normal operating ranges, and to cross some preset alarm limits, thereby generating a set of correlated alarms, which is called an *alarm shower*.

The real-time computer system must detect and display these alarms and must assist the operator in identifying a *primary event* that was the initial cause of these alarms. For this purpose, alarms that are observed must be logged in a special alarm log with the exact instant when the alarm occurred. The exact temporal order of the alarms is helpful in identifying the *secondary alarms*, i.e., all alarms that can be a causal consequence of the primary event. In complex industrial plants, sophisticated knowledge-based systems are used to assist the operator in the alarm analysis.

> **Example:** In the final report on the August 14, 2003 power blackout in the United States and Canada we find on [Tas03, p. 162], the following statement: *A valuable lesson from the August 14 blackout is the importance of having time-synchronized system data recorders. The Task Force's investigators labored over thousands of data items to determine the sequence of events much like putting together small pieces of a very large puzzle. That process would have been significantly faster and easier if there had been wider use of synchronized data recording devices.*

A situation that occurs infrequently but is of utmost concern when it does occur is called a *rare-event* situation. The validation of the performance of a real-time computer system in a rare event situation is a challenging task and requires models of the physical environment (see Sect. 11.3.1).

> **Example:** The sole purpose of a nuclear power plant monitoring and shutdown system is reliable performance in a peak-load alarm situation (a *rare event*). Hopefully, this rare event will never occur during the operational life of the plant.

1.2.2 Direct Digital Control

Many real-time computer systems must calculate the *actuating variables* for the actuators in order to control the controlled object directly (*direct digital control – DDC*), i.e., without any underlying conventional control system.

Control applications are highly regular, consisting of an (infinite) sequence of control cycles, each one starting with sampling (observing) of the RT entities,

followed by the execution of the control algorithm to calculate a new actuating variable, and subsequently by the output of the actuating variable to the actuator. The design of a proper control algorithm that achieves the desired control objective, and compensates for the random disturbances that perturb the controlled object, is the topic of the field of control engineering. In the next section on temporal requirements, some basic notions of control engineering will be introduced.

1.2.3 Man–Machine Interaction

A real-time computer system must inform the operator of the current state of the controlled object, and must assist the operator in controlling the machine or plant object. This is accomplished via the man–machine interface, a critical subsystem of major importance. Many severe computer-related accidents in safety-critical real-time systems have been traced to mistakes made at the man–machine interface [Lev95].

> **Example:** *Mode confusion* at the man–machine interface of an aircraft has been identified to be the cause of major aircraft accidents [Deg95].

Most process-control applications contain, as part of the man–machine interface, an extensive data logging and data reporting subsystem that is designed according to the demands of the particular industry.

> **Example:** In some countries, the pharmaceutical industry is required by law to record and store all relevant process parameters of every production batch in an archival storage in order that the process conditions prevailing at the time of a production run can be reexamined in case a defective product is identified on the market at a later time.

Man–machine interfacing has become such an important issue in the design of computer-based systems that a number of courses dealing with this topic have been developed. In the context of this book, we will introduce an abstract man–machine interface in Sect. 4.5.2, but we will not cover its design in detail. The interested reader is referred to standard textbooks on user interface design.

1.3 Temporal Requirements

1.3.1 Where Do Temporal Requirements Come from?

The most stringent temporal demands for real-time systems have their origin in the requirements of control loops, e.g., in the control of a fast process such as an automotive engine. The temporal requirements at the man–machine interface are, in

comparison, less stringent because the human perception delay, in the range of 50–100 ms, is orders of magnitude larger than the latency requirements of fast control loops.

A Simple Control Loop. Consider the simple control loop depicted in Fig. 1.3 consisting of a vessel with a liquid, a heat exchanger connected to a steam pipe, and a controlling computer system. The objective of the computer system is to control the valve (*control variable*) determining the flow of steam through the heat exchanger such that the temperature of the liquid in the vessel remains within a small range around the *set point* selected by the operator.

The focus of the following discussion is on the temporal properties of this simple control loop consisting of a *controlled object* and a *controlling computer system*.

The Controlled Object. Assume that the system of Fig. 1.3 is in equilibrium. Whenever the steam flow is increased by a step function, the temperature of the liquid in the vessel will change according to Fig. 1.4 until a new equilibrium is reached. This *response function* of the temperature in the vessel depends on the environmental conditions, e.g., the amount of liquid in the vessel, and the flow of steam through the heat exchanger, i.e., on the dynamics of the controlled object. (In the following section, we will use d to denote a *duration* and t to denote an *instant*, i.e., a point in time).

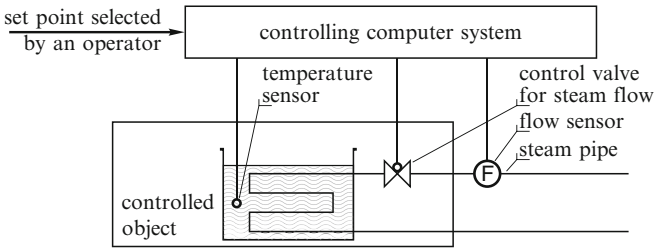

Fig. 1.3 A simple control loop

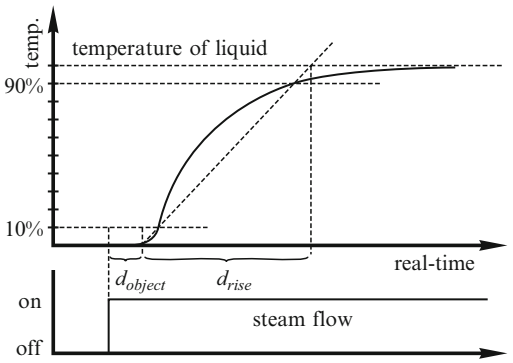

Fig. 1.4 Delay and rise time of the step response

There are two important temporal parameters characterizing this elementary step response function, the *object delay* d^{object} (sometimes called the *lag time* or *lag*) after which the *measured variable* temperature begins to rise (caused by the initial inertia of the process and the instrumentation, called the *process lag*) and the *rise time* d^{rise} of the temperature until the new equilibrium state has been reached. To determine the *object delay* d^{object} and the *rise time* d^{rise} from a given experimentally recorded shape of the step-response function, one finds the two points in time where the response function has reached 10% and 90% of the difference between the two stationary equilibrium values. These two points are connected by a straight line (Fig. 1.4). The significant points in time that characterize the *object delay* d^{object} and the *rise time* d^{rise} of the step response function are constructed by finding the intersection of this straight line with the two horizontal lines that denote the two liquid temperatures that correspond to the stable equilibrium states before and after the application of the step function.

Controlling Computer System. The controlling computer system must sample the temperature of the vessel periodically to detect any deviation between the intended value and the actual value of the controlled variable *temperature*. The constant duration between two sampling points is called the sampling period d^{sample} and the reciprocal $1/d^{sample}$ is the sampling frequency, f^{sample}. A rule of thumb says that, in a digital system which is expected to behave like a quasi-continuous system, the sampling period should be less than one-tenth of the rise time d^{rise} of the step response function of the controlled object, i.e., $d^{sample} < (d^{rise}/10)$. The computer compares the *measured temperature* to the *temperature set point* selected by the operator and calculates the *error term*. This error term forms the basis for the calculation of a new value of the control variable by a *control algorithm*. A given time interval after each sampling point, called the *computer delay* $d^{computer}$, the controlling computer will output this new value of the actuating variable to the control valve, thus closing the control loop. The delay $d^{computer}$ should be smaller than the sampling period d^{sample}.

The difference between the maximum and the minimum values of the delay of the computer is called the *jitter* of the computer delay, $\Delta d^{computer}$. This jitter is a sensitive parameter for the quality of control.

The *dead time* of the control loop is the time interval between the observation of the RT entity and the start of a reaction of the controlled object due to a computer action based on this observation. The dead time is the sum of the controlled object delay d^{object}, which is in the sphere of control of the controlled object and is thus determined by the controlled object's dynamics, and the computer delay $d^{computer}$, which is determined by the computer implementation. To reduce the dead time in a control loop and to improve the stability of the control loop, these delays should be as small as possible. The computer delay $d^{computer}$ is defined by the time interval between the *sampling points,* i.e., the observation of the controlled object, and the *use* of this information (see Fig. 1.5), i.e., the output of the corresponding actuator signal, the *actuating variable*, to the controlled object. Apart from the necessary

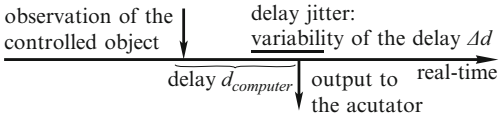

Fig. 1.5 Delay and delay jitter

Table 1.1 Parameters of an elementary control loop

Symbol	Parameter	Sphere of control	Relationships
d^{object}	Controlled object delay	Controlled object	Physical process
d^{rise}	Rise time of step response	Controlled object	Physical process
d^{sample}	Sampling period	Computer	$d^{sample} << d^{rise}$
$d^{computer}$	Computer delay	Computer	$d^{computer} < d^{sample}$
$\Delta d^{computer}$	Jitter of the computer delay	Computer	$\Delta d^{computer} << d^{computer}$
$d^{deadtime}$	Dead time	Computer and controlled object	$d^{computer} + d^{object}$

time for performing the calculations, the computer delay is determined by the time required for communication and the reaction time of the actuator.

Parameters of a Control Loop. Table 1.1 summarizes the temporal parameters that characterize the elementary control loop depicted in Fig. 1.3. In the first two columns we denote the symbol and the name of the parameter. The third column denotes the sphere of control in which the parameter is located, i.e., what subsystem determines the value of the parameter. Finally, the fourth column indicates the relationships between these temporal parameters.

1.3.2 Minimal Latency Jitter

The data items in control applications are state-based, i.e., they contain images of the RT entities. The computational actions in control applications are mostly time-triggered, e.g., the control signal for obtaining a sample is derived from the progression of time within the computer system. This control signal is thus in the sphere of control of the computer system. It is known in advance when the next control action must take place. Many control algorithms are based on the assumption that the delay jitter $\Delta d^{computer}$ is very small compared to the *delay* $d^{computer}$, i.e., the delay is close to constant. This assumption is made because control algorithms can be designed to compensate a *known* constant delay. Delay jitter brings an additional uncertainty into the control loop that has an adverse effect on the quality of control. The jitter Δd can be seen as an uncertainty about the instant when the RT-entity was observed. This jitter can be interpreted as causing an additional value error ΔT of the measured variable temperature T as shown in Fig. 1.6. Therefore, the delay jitter should always be a small fraction of the delay, i.e., if a delay of 1 ms is demanded then the delay jitter should be in the range of a few μs [SAE95].

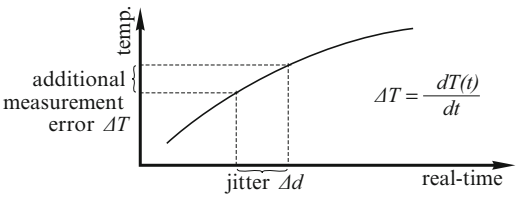

Fig. 1.6 The effect of jitter on the measured variable T

1.3.3 Minimal Error-Detection Latency

Hard real-time applications are, by definition, safety-critical. It is therefore important that any error within the control system, e.g., the loss or corruption of a message or the failure of a node, is detected within a short time with a very high probability. The required *error-detection latency* must be in the same order of magnitude as the sampling period of the fastest critical control loop. It is then possible to perform some corrective action, or to bring the system into a safe state, before the consequences of an error can cause any severe system failure. Almost-no-jitter systems will have shorter guaranteed error-detection latencies than systems that allow for jitter.

1.4 Dependability Requirements

The notion of dependability covers the meta-functional attributes of a computer system that relate to the quality of service a system delivers to its users during an extended interval of time. (A user could be a human or another technical system.) The following measures of dependability attributes are of importance [Avi04]:

1.4.1 Reliability

The *Reliability R(t)* of a system is the probability that a system will provide the specified service until time *t,* given that the system was operational at the beginning, i.e., $t = t_o$. The probability that a system will fail in a given interval of time is expressed by the *failure rate*, measured *in FITs* (Failure In Time). A failure rate of 1 *FIT* means that the mean time to a failure *(MTTF)* of a device is 10^9 h, i.e., one failure occurs in about 115,000 years. If a system has a constant *failure rate* of λ *failures*/*h*, then the reliability at time t is given by

$$R(t) = \exp(-\lambda(t - t_o)),$$

where $t - t_o$ is given in hours. The inverse of the failure rate $1/\lambda = MTTF$ is called the *Mean-Time-To-Failure MTTF* (in hours). If the failure rate of a system is required to be in the order of 10^{-9} failures/h or lower, then we speak of a system with an *ultrahigh reliability* requirement.

1.4.2 Safety

Safety is reliability regarding *critical failure modes*. A critical failure mode is said to be *malign*, in contrast with a noncritical failure, which is *benign*. In a malign failure mode, the cost of a failure can be orders of magnitude higher than the utility of the system during normal operation. Examples of malign failures are: an airplane crash due to a failure in the flight-control system, and an automobile accident due to a failure of a computer-controlled intelligent brake in the automobile. Safety-critical (hard) real-time systems must have a failure rate with regard to critical failure modes that conforms to the *ultrahigh reliability* requirement.

> **Example:** Consider the example of a computer-controlled brake in an automobile. The failure rate of a computer-caused critical brake failure must be lower than the failure rate of a conventional braking system. Under the assumption that a car is operated about 1 h per day on the average, one safety-critical failure per million cars per year translates into a failure rate in the order of 10^{-9} failures/h.

Similarly low failure rates are required in flight-control systems, train-signaling systems, and nuclear power plant monitoring systems.

Certification. In many cases the design of a safety-critical real-time system must be approved by an independent certification agency. The certification process can be simplified if the certification agency can be convinced that:

1. The subsystems that are critical for the safe operation of the system are protected by fault-containment mechanisms that eliminate the possibility of error propagation from the rest of the system into these safety-critical subsystems.
2. From the point of view of design, all scenarios that are covered by the given load- and fault-hypothesis can be handled according to the specification without reference to probabilistic arguments. This makes a resource adequate design necessary.
3. The architecture supports a constructive modular certification process where the certification of subsystems can be done independently of each other. At the system level, only the *emergent properties* must be validated.

[Joh92] specifies the required properties for a system that is *designed for validation*:

1. A complete and accurate reliability model can be constructed. All parameters of the model that cannot be deduced analytically must be measurable in feasible time under test.

2. The reliability model does not include state transitions representing design faults; analytical arguments must be presented to show that design faults will not cause system failure.
3. Design tradeoffs are made in favor of designs that minimize the number of parameters that must be measured (see Sect. 2.2.1).

1.4.3 Maintainability

Maintainability is a measure of the time interval required to repair a system after the occurrence of a benign failure. Maintainability is measured by the probability *M (d)* that the system is restored within a time interval *d* after the failure. In keeping with the reliability formalism, a constant repair rate μ (repairs per hour) and a *Mean Time to Repair (MTTR)* are introduced to define a quantitative maintainability measure.

There is a fundamental conflict between reliability and maintainability. A maintainable design requires the partitioning of a system into a set of *field replaceable units* (*FRUs*) connected by serviceable interfaces that can be easily disconnected and reconnected to replace a faulty *FRU* in case of a failure. A serviceable interface, e.g., a plug connection, has a significantly higher physical failure rate than a non-serviceable interface. Furthermore, a serviceable interface is more expensive to produce.

In the upcoming field of *ambient intelligence,* automatic diagnosis and *maintainability by an untrained end user* will be important system properties that are critical for the market success of a product.

1.4.4 Availability

Availability is a measure of the delivery of correct service with respect to the alternation of correct and incorrect service. It is measured by the fraction of time that the system is ready to provide the service.

> **Example:** Whenever a user picks up the phone, the telephone switching system should be ready to provide the telephone service with a very high probability. A telephone exchange is allowed to be out of service for only a few minutes per year.

In systems with constant failure and repair rates, the reliability *(MTTF),* maintainability *(MTTR)*, and availability *(A)* measures are related by

$$A = MTTF/(MTTF + MTTR).$$

The sum $MTTF + MTTR$ is sometimes called the *Mean Time Between Failures (MTBF)*. Figure 1.7 shows the relationship between *MTTF, MTTR, and MTBF*.

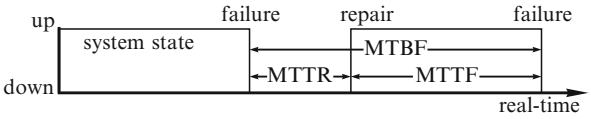

Fig. 1.7 Relationship between *MTTF*, *MTBF* and *MTTR*

A high availability can be achieved either by a long *MTTF* or by a short *MTTR*. The designer has thus some freedom in the selection of her/his approach to the construction of a high-availability system.

1.4.5 Security

A fifth important attribute of dependability – the *security attribute* – is concerned with the authenticity and integrity of information, and the ability of a system to prevent unauthorized access to information or services (see also Sect. 6.2). There are difficulties in defining a quantitative security measure, e.g., the specification of a *standard burglar* that takes a certain time to intrude a system. Traditionally, security issues have been associated with large databases, where the concerns are confidentiality, privacy, and authenticity of information. During the last few years, security issues have also become important in real-time systems, e.g., a cryptographic theft-avoidance system that locks the ignition of a car if the user cannot present the specified access code. In the *Internet-of-Things (IoT)*, where the endpoints of the Internet are embedded systems that bridge the gap between the cyber world and physical word, security concerns are of crucial importance, since an intruder cannot only corrupt a data structure in a computer, but can cause harm in the physical environment.

1.5 Classification of Real-Time Systems

In this section we classify real-time systems from different perspectives. The first two classifications, *hard real-time* versus *soft real-time* (on-line), and *fail-safe* versus *fail-operational*, depend on the characteristics of the application, i.e., on factors *outside* the computer system. The second three classifications, *guaranteed-timeliness* versus *best effort*, *resource-adequate* versus *resource-inadequate*, and *event-triggered* versus *time-triggered*, depend on the design and implementation of the computer application, i.e., on factors *inside* the computer system.

1.5.1 Hard Real-Time System Versus Soft Real-Time System

The design of a *hard real-time system*, which always must produce the results at the correct instant, is fundamentally different from the design of a *soft-real time* or an *on-line system*, such as a transaction processing system. In this section we will

Table 1.2 Hard real-time versus soft real-time systems

Characteristic	Hard real-time	Soft real-time (on-line)
Response time	Hard-required	Soft-desired
Peak-load performance	Predictable	Degraded
Control of pace	Environment	Computer
Safety	Often critical	Non-critical
Size of data files	Small/medium	Large
Redundancy type	Active	Checkpoint–recovery
Data integrity	Short-term	Long-term
Error detection	Autonomous	User assisted

elaborate on these differences. Table 1.2 compares the characteristics of hard real-time systems versus soft real-time systems.

Response Time. The demanding response time requirements of hard real-time applications, often in the order of milliseconds or less, preclude direct human intervention during normal operation or in critical situations. A hard real-time system must be highly autonomous to maintain safe operation of the process. In contrast, the response time requirements of soft real-time and on-line systems are often in the order of seconds. Furthermore, if a deadline is missed in a soft real-time system, no catastrophe can result.

Peak-load Performance. In a hard real-time system, the peak-load scenario must be well defined. It must be guaranteed by design that the computer system meets the specified deadlines in all situations, since the utility of many hard real-time applications depends on their predictable performance during *rare event scenarios* leading to a peak load. This is in contrast to the situation in a soft-real time system, where the *average* performance is important, and a degraded operation in a rarely occurring peak load case is tolerated for economic reasons.

Control of Pace. A hard real-time computer system is often paced by the state changes occurring in the environment. It must keep up with the state of the environment (the controlled object and the human operator) under all circumstances. This is in contrast to an on-line system, which can exercise some control over the environment in case it cannot process the offered load.

> **Example:** Consider the case of a transaction processing system, such as an airline reservation system. If the computer cannot keep up with the demands of the users, it just extends the response time and forces the users to slow down.

Safety. The safety criticality of many real-time applications has a number of consequences for the system designer. In particular, error detection and recovery must be autonomous such that the system can initiate appropriate recovery actions and arrive at a safe state within the time intervals dictated by the application without human intervention.

Size of Data Files. The *real-time database* that is composed of the temporally accurate images of the RT-entities is normally of small size. The key concern in

hard real-time systems is on the *short-term* temporal accuracy of the real-time database that is invalidated by the flow of real-time. In contrast, in on-line transaction processing systems, the maintenance of the *long-term* integrity and availability of large data files is the key issue.

Redundancy Type. After an error has been detected in an on-line system, the computation is rolled back to a previously established checkpoint to initiate a recovery action. In hard real-time systems, roll-back/recovery is of limited utility for the following reasons:

1. It is difficult to guarantee the deadline after the occurrence of an error, since the roll-back/recovery action can take an unpredictable amount of time.
2. An irrevocable action that has been effected on the environment cannot be undone.
3. The temporal accuracy of the checkpoint data may be invalidated by the time difference between the checkpoint time and the instant *now*.

The topic of temporal accuracy of real-time data is discussed at length in Sect. 5.4 while the issues of error detection and types of redundancy are dealt with in Chap. 6.

1.5.2 Fail-Safe Versus Fail-Operational

In many hard real-time systems one or more safe states, which can be reached in case of a system failure, can be identified. If such a safe state can be identified and quickly reached upon the occurrence of a failure, then we call the system *fail-safe*. Fail-safeness is a characteristic of the controlled object, not the computer system. In fail-safe applications the computer system must have a *high error-detection coverage*, i.e., the probability that an error is detected, provided it has occurred, must be close to one.

> **Example:** In case a failure is detected in a railway signaling system, it is possible to set all signals to red and thus stop all the trains in order to bring the system to a safe state.

In many real-time computer systems a special external device, a *watchdog*, is provided to monitor the operation of the computer system. The computer system must send a periodic life-sign (e.g., a digital output of predefined form) to the watchdog. If this life-sign fails to arrive at the watchdog within the specified time interval, the watchdog assumes that the computer system has failed and forces the controlled object into a safe state. In such a system, timeliness is needed only to achieve high availability, but is not needed to maintain safety since the watchdog forces the controlled object into a safe state in case of a timing violation.

There are, however, applications where a safe state cannot be identified, e.g., a flight control system aboard an airplane. In such an application the computer system must remain operational and provide a minimal level of service even in the case of a failure to avoid a catastrophe. This is why these applications are called *fail-operational*.

1.5.3 Guaranteed-Response Versus Best-Effort

If we start out with a specified *fault-* and *load-hypothesis* and deliver a design that makes it possible to reason about the adequacy of the design without reference to probabilistic arguments (even in the case of a peak load and fault scenario), then we can speak of a system with a *guaranteed response*. The probability of failure of a perfect system with guaranteed response is reduced to the probability that the assumptions about the peak load and the number and types of faults do not hold in reality. This probability is called *assumption coverage* [Pow95]. Guaranteed response systems require careful planning and extensive analysis during the design phase.

If such an analytic response guarantee cannot be given, we speak of a *best-effort* design. Best-effort systems do not require a rigorous specification of the load- and fault-hypothesis. The design proceeds according to the principle *best possible effort taken* and the sufficiency of the design is established during the test and integration phases. It is difficult to establish that a best-effort design operates correctly in a rare-event scenario. At present, many non safety-critical real-time systems are designed according to the best-effort paradigm.

1.5.4 Resource-Adequate Versus Resource-Inadequate

Guaranteed response systems are based on the principle of resource adequacy, i.e., there are enough computing resources available to handle the specified peak load and the fault scenario. Many non safety-critical real-time system designs are based on the principle of resource inadequacy. It is assumed that the provision of sufficient resources to handle every possible situation is not economically viable, and that a dynamic resource allocation strategy based on resource sharing and probabilistic arguments about the expected load and fault scenarios is acceptable.

It is expected that, in the future, there will be a paradigm shift to resource-adequate designs in many applications. The use of computers in important volume-based applications, e.g., in cars, raises both the public awareness as well as concerns about computer-related incidents, and forces the designer to provide convincing arguments that the design functions properly under *all* stated conditions. Hard real-time systems must be designed according to the guaranteed response paradigm that requires the availability of adequate resources.

1.5.5 Event-Triggered Versus Time-Triggered

The distinction between *event-triggered* and *time-triggered* depends on the type of internal *triggers* and not the external behavior of a real-time system. A *trigger* is an event that causes the start of some action in the computer, e.g., the execution of a

task or the transmission of a message. Depending on the triggering mechanisms for the start of communication and processing actions in each node of a computer system, two distinctly different approaches to the design of the control mechanisms of real-time computer applications can be identified, *event-triggered control* and *time-triggered control*.

In *event-triggered (ET)* control, all communication and processing activities are initiated whenever a significant event other than the regular event of a clock tick occurs. In an ET system, the signaling of significant events to the central processing unit (CPU) of a computer is realized by the well-known interrupt mechanism. ET systems require a dynamic scheduling strategy to activate the appropriate software task that services the event.

In a *time-triggered (TT) system*, all activities are initiated by the progression of real-time. There is only one interrupt in each node of a distributed TT system, the periodic real-time clock interrupt. Every communication or processing activity is initiated at a periodically occurring predetermined tick of a clock. In a distributed TT real-time system, it is assumed that the clocks of all nodes are synchronized to form a *global time* that is available at every node. Every observation of the controlled object is time-stamped with this global time. The granularity of the global time must be chosen such that the time order of any two observations made anywhere in a distributed TT system can be established from their time-stamps with adequate faithfulness [Kop09]. The topics of global time and clock synchronization will be discussed at length in Chap. 3.

Example: The distinction between *event-triggered* and *time-triggered* can be explained by an example of an elevator control system. When you push a *call button* in the event-triggered implementation, the event is immediately relayed to the interrupt system of the computer in order to start the action of calling the elevator. In a time-triggered system, the button push is stored locally, and periodically, e.g., every second, the computer asks to get the state of all push buttons. The flow of control in a time-triggered system is managed by the progression of time, while in an event-triggered system; the flow of control is determined by the events that happen in the environment or the computer system.

1.6 The Real-Time Systems Market

In a market economy, the cost/performance relation is a decisive parameter for the market success of any product. There are only a few scenarios where cost arguments are not the major concern. The total life-cycle cost of a product can be broken down into three rough categories: non-recurring development cost, production cost, and operation and maintenance cost. Depending on the product type, the distribution of the total life-cycle cost over these three cost categories can vary significantly. We will examine this life cycle cost distribution by looking at two important examples of real-time systems, embedded real-time systems and plant-automation systems.

1.6.1 Embedded Real-Time Systems

The ever-decreasing price/performance ratio of microcontrollers makes it economically attractive to replace conventional mechanical or electronic control system within many products by an embedded real-time computer system. There are numerous examples of products with embedded computer systems: cellular phones, engine controllers in cars, heart pacemakers, computer printers, television sets, washing machines, even some electric razors contain a microcontroller with some thousand instructions of software code. Because the external interfaces (particularly the man–machine interface) of the product often remain unchanged relative to the previous product generation, it is often not visible from the outside that a real-time computer system is controlling the product behavior.

Characteristics. An embedded real-time computer system is always part of a well-specified larger system, which we call an *intelligent product*. An intelligent product consists of a physical (mechanical) subsystem; the controlling embedded computer, and, most often, a man–machine interface. The ultimate success of any intelligent product depends on the relevance and quality of service it can provide to its users. A focus on the genuine user needs is thus of utmost importance.

Embedded systems have a number of distinctive characteristics that influence the system development process:

1. Mass Production: many embedded systems are designed for a mass market and consequently for mass production in highly automated assembly plants. This implies that the production cost of a single unit must be as low as possible, i.e., efficient memory and processor utilization are of concern.
2. Static Structure: the computer system is embedded in an intelligent product of given functionality and rigid structure. The known *a priori* static environment can be analyzed at design time to simplify the software, to increase the robustness, and to improve the efficiency of the embedded computer system. In many embedded systems there is no need for flexible dynamic software mechanisms. These mechanisms increase the resource requirements and lead to an unnecessary complexity of the implementation.
3. Man–Machine Interface: if an embedded system has a man–machine interface, it must be specifically designed for the stated purpose and must be easy to operate. Ideally, the use of the intelligent product should be self-explanatory, and not require any training or reference to an operating manual.
4. Minimization of the Mechanical Subsystem: to reduce the manufacturing cost and to increase the reliability of the intelligent product, the complexity of the mechanical subsystem is minimized.
5. Functionality Determined by Software in Read-Only Memory (ROM): the integrated software that often resides in ROM determines the functionality of many intelligent products. Since it is not possible to modify the software in a ROM after its release, the quality standards for this software are high.

6. Maintenance Strategy: many intelligent products are designed to be non maintainable, because the partitioning of the product into replaceable units is too expensive. If, however, a product is designed to be maintained in the field, the provision of an excellent diagnostic interface and a self-evident maintenance strategy is of importance.
7. Ability to communicate: many intelligent products are required to interconnect with some larger system or the Internet. Whenever a connection to the Internet is supported, the topic of security is of utmost concern.
8. Limited amount of energy: Many mobile embedded devices are powered by a battery. The lifetime of a battery load is a critical parameter for the utility of a system.

A large fraction of the life-cycle cost of many intelligent products is in the production, i.e., in the hardware. The known *a priori* static configuration of the intelligent product can be used to reduce the resource requirements, and thus the production cost, and also to increase the robustness of the embedded computer system. Maintenance cost can become significant, particularly if an undetected design fault (software fault) requires a recall of the product, and the replacement of a complete production series.

> **Example:** In [Neu96] we find the following laconic one-liner: *General Motors recalls almost 300 K cars for engine software flaw.*

Future Trends. During the last few years, the variety and number of embedded computer applications have grown to the point that, by now, this segment is by far the most important one in the computer market. The embedded system market is driven by the continuing improvements in the cost/performance ratio of the semiconductor industry that makes computer-based control systems cost-competitive relative to their mechanical, hydraulic, and electronic counterparts. Among the key mass markets are the domains of consumer electronics and automotive electronics. The automotive electronics market is of particular interest, because of stringent timing, dependability, and cost requirements that act as *technology catalysts*.

Automotive manufacturers view the proper exploitation of computer technology as a key competitive element in the never-ending quest for increased vehicle performance and reduced manufacturing cost. While some years ago, the computer applications on board a car focused on non-critical body electronics or comfort functions, there is now a substantial growth in the computer control of core vehicle functions, e.g., engine control, brake control, transmission control, and suspension control. We observe an integration of many of these functions with the goal of increasing the vehicle stability in critical driving maneuvers. Obviously, an error in any of these core vehicle functions has severe safety implications.

At present the topic of computer safety in cars is approached at two levels. At the basic level a mechanical system provides the proven safety level that is considered sufficient to operate the car. The computer system provides optimized performance on top of the basic mechanical system. In case the computer system fails cleanly, the mechanical system takes over. Consider, for example, an *Electronic Stability*

Program (ESP). If the computer fails, the conventional mechanical brake system is still operational. Soon, this approach to safety may reach its limits for two reasons:

1. If the computer-controlled system is further improved, the magnitude of the difference between the performance of the computer-controlled system and the performance of the basic mechanical system is further increased. A driver who is used to the high performance of the computer-controlled system might consider the fallback to the inferior performance of the mechanical system a safety risk.
2. The improved price/performance of the microelectronic devices will make the implementation of fault-tolerant computer systems cheaper than the implementation of mixed computer/mechanical systems. Thus, there will be an economic pressure to eliminate the redundant mechanical system and to replace it with a computer system using active redundancy.

The embedded system market is expected to grow significantly during the next 10 years. It is expected that many embedded systems will be connected to the Internet, forming the Internet of Things (IoT – see Chap. 13).

1.6.2 Plant Automation Systems

Characteristics. Historically, industrial plant automation was the first field for the application of real-time digital computer control. This is understandable since the benefits that can be gained by the computerization of a sizable plant are much larger than the cost of even an expensive process control computer of the late 1960s. In the early days, human operators controlled the industrial plants locally. With the refinement of industrial plant instrumentation and the availability of remote automatic controllers, plant monitoring and command facilities were concentrated into a central control room, thus reducing the number of operators required to run the plant. In the 1970s, the next logical step was the introduction of central process control computers to monitor the plant and assist the operator in her/his routine functions, e.g., data logging and operator guidance. In the beginning, the computer was considered an *add-on* facility that was not fully trusted. It was the duty of the operator to judge whether a set point calculated by a computer made sense and could be applied to the process (*open-loop control*). In the next phase, *Supervisory Control and Data Acquisition* (SCADA) systems calculated the set-points for the *programmable logic controllers* (PLC) in the plant. With the improvement of the process models and the growth of the reliability of the computer, control functions have been increasingly allocated to the computer and gradually the operator has been taken out of the control loop (*closed-loop control*). Sophisticated non-linear control techniques, which have response time requirements beyond human capabilities, have been implemented.

Usually, every plant automation system is unique. There is an extensive amount of engineering and software effort required to adapt the computer system to the

physical layout, the operating strategy, the rules and regulations, and the reporting system of a particular plant. To reduce these engineering and software efforts, many process control companies have developed a set of modular building blocks, which can be configured individually to meet the requirements of a customer. Compared to the development cost, the production cost (hardware cost) is of minor importance. Maintenance cost can be an issue if a maintenance technician must be on-site for 24 h in order to minimize the downtime of a plant.

Future Trends. The market of industrial plant automation systems is limited by the number of plants that are newly constructed or are refurbished to install a computer control system. During the last 20 years, many plants have already been automated. This investment must pay off before a new generation of computers and control equipment is installed.

Furthermore, the installation of a new generation of control equipment in a production plant causes disruption in the operation of the plant with a costly loss of production that must be justified economically. This is difficult if the plant's efficiency is already high, and the margin for further improvement by refined computer control is limited.

The size of the plant automation market is too small to support the mass production of special application-specific components. This is the reason why many VLSI components that are developed for other application domains, such as automotive electronics, are taken up by this market to reduce the system cost. Examples of such components are sensors, actuators, real-time local area networks, and processing nodes. Already several process-control companies have announced a new generation of process-control equipment that takes advantage the of low-priced mass produced components that have been developed for the automotive market, such as the chips developed for the Controller Area Network (CAN – see Sect. 7.3.2).

1.6.3 Multimedia Systems

Characteristics. The multimedia market is a mass market for specially designed soft and firm real-time systems. Although the deadlines for many multimedia tasks, such as the synchronization of audio and video streams, are firm, they are not hard deadlines. An occasional failure to meet a deadline results in a degradation of the *quality of the user experience*, but will not cause a catastrophe. The processing power required to transport and render a continuous video stream is large and difficult to estimate, because it is possible to improve a good picture even further. The resource allocation strategy in multimedia applications is thus quite different from that of hard real-time applications; it is not determined by the given application requirements, but by the amount of available resources. A fraction of the given computational resources (processing power, memory, bandwidth) is allocated to a user domain. *Quality of experience* considerations at the end user determine the

detailed resource allocation strategy. For example, if a user reduces the size of a window and enlarges the size of another window on his multimedia terminal, then the system can reduce the bandwidth and the processing allocated to the first window to free the resources for the other window that has been enlarged. Other users of the system should not be affected by this local reallocation of resources.

Future Trends. The marriage of the Internet with smart phones and multimedia personal computers leads to many new volume applications. The focus of this book is not on multimedia systems, because these systems belong to the class of soft and firm real-time applications.

1.7 Examples of Real-Time Systems

In this section, three typical examples of real-time systems are introduced. These examples will be used throughout the book to explain the evolving concepts. We start with an example of a very simple system for flow control to demonstrate the need for *end-to-end protocols* in process input/output.

1.7.1 Controlling the Flow in a Pipe

It is the objective of the simple control system depicted in Fig. 1.8 to control the flow of a liquid in a pipe. A given flow set point determined by a client should be maintained despite changing environmental conditions. Examples for such changing conditions are the varying level of the liquid in the vessel or the temperature sensitive viscosity of the liquid. The computer interacts with the controlled object by setting the position of the control valve. It then observes the reaction of the controlled object by reading the flow sensor F to determine whether the desired effect, the intended change of flow, has been achieved. This is a typical example of the necessary *end-to-end protocol* [Sal84] that must be put in place between the computer and the controlled object (see also Sect. 7.1.2). In a well-engineered system, the effect of any control action of the computer must be monitored by one or more independent sensors. For this purpose, many actuators contain a number of sensors in the same physical housing. For example, the control valve in Fig. 1.8 might contain a sensor, which measures the mechanical position of the valve in the

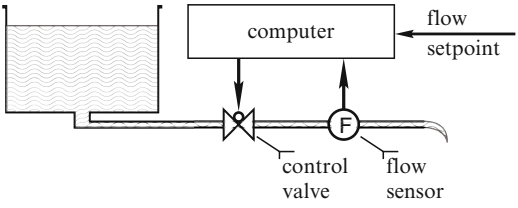

Fig. 1.8 Flow of liquid in a pipe

pipe, and two limit switches, which indicate the firmly closed and the completely open positions of the valve. A rule of thumb is that there are about three to seven sensors for every actuator.

The dynamics of the system in Fig. 1.8 is essentially determined by the speed of the control valve. Assume that the control valve takes 10 s to open or close from 0% to 100%, and that the flow sensor F has a precision of 1%. If a sampling interval of 100 ms is chosen, the maximum change of the valve position within one sampling interval is 1%, the same as the precision of the flow sensor. Because of this finite speed of the control valve, an output action taken by the computer at a given time will lead to an effect in the environment at some later time. The observation of this effect by the computer will be further delayed by the given latency of the sensor. All these latencies must either be derived analytically or measured experimentally, before the temporal control structure for a stable control system can be designed.

1.7.2 Engine Control

The task of an engine controller in an automobile engine is the calculation of the proper amount of fuel and the exact moment at which the fuel must be injected into the combustion chamber of each cylinder. The amount of fuel and the timing depend on a multitude of parameters: the intentions of the driver, articulated by the position of the accelerator pedal, the current load on the engine, the temperature of the engine, the condition of the cylinder, and many more. A modern engine controller is a complex piece of equipment. Up to 100 concurrently executing software tasks must cooperate in tight synchronization to achieve the desired goal, a smoothly running and efficient engine with a minimal output of pollutants.

The up- and downward moving piston in each cylinder of a combustion engine is connected to a rotating axle, the *crankshaft*. The intended start point of fuel injection is relative to the position of the piston in the cylinder, and must be precise within an accuracy of about 0.1° of the measured angular position of the crankshaft. The precise angular position of the crankshaft is measured by a number of digital sensors that generate a rising edge of a signal at the instant when the crankshaft passes these defined positions. Consider an engine that turns with 6,000 rpm (revolutions per minute), i.e., the crankshaft takes 10 ms for a 360° rotation. If the required precision of 0.1° is transformed into the time domain, then a temporal accuracy of 3 μs is required. The fuel injection is realized by opening a solenoid valve or a piezoelectric actuator that controls the fuel flow from a high-pressure reservoir into the cylinder. The latency between giving an *open* command to the valve and the actual point in time when the valve opens can be in the order of hundreds of μs, and changes considerably depending on environmental conditions (e.g., temperature). To be able to compensate for this latency jitter, a sensor signal indicates the point in time when the valve has actually opened. The duration between the execution of the output command by the computer and the start of opening of the valve is measured during every engine cycle. The measured latency

is used to determine when the output command must be executed during the next cycle so that the intended effect, the start of fuel injection, happens at the proper point in time.

This example of an engine controller has been chosen because it demonstrates convincingly the need for extremely precise temporal control. For example, if the processing of the signal that measures the exact position of the crankshaft in the engine is delayed by a few μs, the quality of control of the whole system is compromised. It can even happen that the engine is mechanically damaged if a valve is opened at an incorrect moment.

1.7.3 Rolling Mill

A typical example of a distributed plant automation system is the computer control of a rolling mill. In this application a slab of steel (or some other material, such as paper) is rolled to a strip and coiled. The rolling mill of Fig. 1.9 has three drives and some instrumentation to measure the quality of the rolled product. The distributed computer-control system of this rolling mill consists of seven nodes connected by a real-time communication system. The most important sequence of actions – we call this a *real-time (RT) transaction* – in this application starts with the reading of the sensor values by the sensor computer. Then, the RT transaction passes through the model computer that calculates new set points for the three drives, and finally reaches the control computers to achieve the desired action by readjusting the rolls of the mill. The RT-transaction thus consists of three processing actions connected by two communication actions.

The total duration of the RT transaction (bold line in Fig. 1.9) is an important parameter for the quality of control. The shorter the duration of this transaction, the better the control quality and the stability of the control loop, since this transaction contributes to the *dead time* of the critical control loop. The other important term of the dead time is the time it takes for the strip to travel from the drive to the sensor. A jitter in the dead time that is not compensated for will reduce the quality of

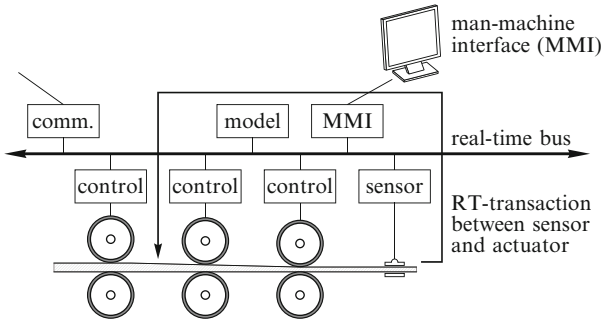

Fig. 1.9 An RT transaction

control significantly. It is evident from Fig. 1.9 that the latency jitter in an event-triggered system is the sum of the jitter of all processing and communication actions that form the critical RT transaction.

Note that the communication pattern among the nodes of this control system is *multicast*, not *point-to-point*. This is typical for most distributed real-time control systems. Furthermore, the communication between the model node and the drive nodes has an *atomicity requirement*. Either all of the drives are changed according to the output of the model, or none of them is changed. The loss of a message, which may result in the failure of a drive to readjust to a new position, may cause mechanical damage to the drive.

Points to Remember

- A real-time computer system must react to stimuli from the controlled object (or the operator) within time intervals *dictated* by its environment. If a catastrophe could result in case a firm deadline is missed, the deadline is called *hard*.
- In a hard real-time computer system, it must be guaranteed by design that the computer system will meet the specified deadlines in all situations because the utility of many hard real-time applications can depend on predictable performance during a peak load scenario.
- A hard real-time system must maintain synchrony with the state of the environment (the controlled object and the human operator) in all operational scenarios. It is thus paced by the state changes occurring in the environment.
- Because the state of the controlled object changes as a function of real-time, an observation is *temporally accurate* only for a limited time interval.
- A *trigger* is an event that causes the start of some action, e.g., the execution of a task or the transmission of a message.
- Real-time systems have only small data files, the *real-time database* that is formed by the temporally accurate images of the RT-entities. The key concern is on the *short-term* temporal accuracy of the real-time database that is invalidated by the flow of real-time.
- The real-time database must be updated whenever an RT entity changes its value. This update can be performed periodically, triggered by the progression of the real-time clock by a fixed period (*time-triggered observation*), or immediately after the occurrence of an event in the RT entity (*event-triggered observation*).
- The most stringent temporal demands for real-time systems have their origin in the requirements of the control loops.
- The temporal behavior of a simple controlled object can be characterized by *process lag* and *rise time* of the *step-response function*.
- The *dead time* of a control loop is the time interval between the observation of the RT entity and the start of a reaction of the controlled object as a consequence of a computer action based on this observation.
- Many control algorithms are based on the assumption that the delay jitter is a very small fraction of the *delay* since control algorithms are designed to

compensate a known constant delay. Delay jitter brings an additional uncertainty into the control loop that has an adverse effect on the quality of control.

- The term *signal conditioning* is used to refer to all processing steps that are needed to get a meaningful RT image of an RT entity from the raw sensor data.
- The *Reliability R(t)* of a system is the probability that a system will provide the specified service until time t, given that the system was operational at $t = t_o$.
- If the failure rate of a system is required to be about 10^{-9} failures/h or lower, then we are dealing with a system with an *ultrahigh reliability* requirement.
- Safety is reliability regarding *malign (critical) failure modes*. In a malign failure mode, the cost of a failure can be orders of magnitude higher than the utility of the system during normal operation.
- *Maintainability* is a measure of the time it takes to repair a system after the last experienced benign failure, and is measured by the probability *M(d)* that the system is restored within a time interval d after the failure.
- *Availability* is a measure for the correct service delivery regarding the alternation of correct and incorrect service, and is measured by the probability *A(t)* that the system is ready to provide the service at time t.
- The main security concerns in real-time systems are the *authenticity*, *integrity*, and *timeliness* of the real-time information.
- The probability of failure of a perfect system with guaranteed response is reduced to the probability that the assumptions concerning the peak load and the number and types of faults are valid in reality.
- If we start out from a specified fault- and load-hypothesis and deliver a design that makes it possible to reason about the adequacy of the design without reference to probabilistic arguments (even in the case of the extreme load and fault scenarios) we speak of a system with a *guaranteed response*.
- An embedded real-time computer system is part of a well-specified larger system, an *intelligent product*. An intelligent product normally consists of a mechanical subsystem, the controlling embedded computer, and a man–machine interface.
- The static configuration, known *a priori*, of the intelligent product can be used to reduce the resource requirements and increase the robustness of the embedded computer system.
- Usually, every plant automation system is unique. Compared to development cost, the production cost (hardware cost) of a plant automation system is less important.
- The embedded system market is expected to grow significantly during the next 10 years. Compared with other information technology markets, this market will offer the best employment opportunities for the computer engineers of the future.

Bibliographic Notes

There exist a number of textbooks on real-time and embedded systems, such as *Introduction to Embedded Systems – A Cyber-Physical Systems Approach* [Lee10] by Ed Lee and Seshia, *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems* [Mar10] by Peter Marwedel, *Real-Time Systems* by

Jane Liu [Liu00], *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications* by Giorgio Buttazzo [But04], and *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX* by Burns and Wellings [Bur09]. The Journal *Real-Time Systems* by Springer publishes archival research articles on the topic.

Review Questions and Problems

1.1 What makes a computer system a *real-time* computer system?
1.2 What are typical functions that a real-time computer system must perform?
1.4 Where do the *temporal requirements* come from? What are the parameters that describe the temporal characteristics of a controlled object?
1.5 Give a "rule of thumb" that relates the *sampling period* in a quasi-continuous system to the *rise time* of the step-response function of the controlled object.
1.6 What are the effects of delay and delay jitter on the quality of control? Compare the error-detection latency in systems with and without jitter.
1.7 What does *signal conditioning* mean?
1.8 Consider an RT entity that changes its value periodically according to $v(t) = A_o \sin(2\pi t/T)$ where T, the period of the oscillation, is 100 ms. What is the maximum change of value of this RT entity within a time interval of 1 ms? (express the result in percentage of the amplitude A_o).
1.9 Consider an engine that rotates with 3,000 rpm. By how many degrees will the crankshaft turn within 1 ms?
1.10 Give some examples where the predictable rare-event performance determines the utility of a hard real-time system.
1.11 Consider a fail-safe application. Is it necessary that the computer system provides guaranteed timeliness to maintain the safety of the application? What is the level of error-detection coverage required in an ultrahigh dependability application?
1.12 What is the difference between availability and reliability? What is the relationship between maintainability and reliability?
1.13 When is there a simple relation between the MTTF and the failure rate?
1.14 Assume you are asked to certify a safety-critical control system. How would you proceed?
1.15 What are the main differences between a soft real-time system and a hard real-time system?
1.16 Why is an *end-to-end protocol* required at the interface between the computer system and the controlled object?
1.17 What is the fraction *development cost*/*production cost* in embedded systems and in plant automation systems? How does this relation influence the system design?
1.19 Assume that an automotive company produces 2,000,000 electronic engine controllers of a special type. The following design alternatives are discussed:

(a) Construct the engine control unit as a single SRU with the application software in Read Only Memory (ROM).The production cost of such a unit is $250. In case of an error, the complete unit has to be replaced.
(b) Construct the engine control unit such that the software is contained in a ROM that is placed on a socket and can be replaced in case of a software error. The production cost of the unit without the ROM is $248. The cost of the ROM is $5.
(c) Construct the engine control unit as a single SRU where the software is loaded in a Flash EPROM that can be reloaded. The production cost of such a unit is $255.

The labor cost of repair is assumed to be $50 for each vehicle. (It is assumed to be the same for each one of the three alternatives). Calculate the cost of a software error for each one of the three alternative designs if 300,000 cars have to be recalled because of the software error (example in Sect. 1.6.1). Which one is the lowest cost alternative if only 1,000 cars are affected by a recall?

1.20 Estimate the relation (development cost)/(production cost) in an embedded consumer application and in a plant automation system.

1.21 Compare the peak load (number of messages, number of task activations inside the computer) that can be generated in an event-triggered and a time-triggered implementation of an elevator control system!