

Chapter 1

Real Time System Characteristics

Real-Time systems are designed to cater to many applications ranging from simple home appliances and laboratory instruments to complex control systems for chemical and nuclear plants, flight guidance of aircrafts and ballistic missiles. All these applications require a computational system (including both the computer and software) interacting with physical equipments like sensors and actuators. Such systems are often referred to as *embedded systems*.

An important feature of many of these systems is the ability to provide *continual* and *timely* response to unpredictable changes in the state of the environment. Hence, these systems have relatively rigid performance requirements. Further, these systems have to satisfy stringent *fail-safe* reliability requirements as failure in many of the applications will result in economic, human or ecological catastrophes. For these reasons, these systems are called *safety-critical* or *time-critical* systems.

In general, the interface between a real-time system and its environment tends to be complex, asynchronous, and distributed. This is due to the fact that the environment of the system consists of a number of physical entities that have autonomous behavior and that interact with the systems asynchronously; it is probably the complexity of the environment that necessitates computer support in the first place. Such systems can be extraordinarily hard to test. The complexity of the environment interface is one obstacle, and the fact that these programs often cannot be tested in their operational environments is another. It is not feasible to test flight-guidance software by flying with it, nor to test ballistic-missile-defense software un-

der battle conditions. In summary, some of the important characteristics of real-time systems are:

- The environment that a system interacts with, is highly nondeterministic and often consists of asynchronous distributed units; there is no way to anticipate in advance the precise order of different external events.
- High speed external events may affect the flow of control in the system easily.
- Responses to external events should be within strict bounded time limits.
- They tend to be large, complex and extraordinarily hard to test.
- In some real time applications, the mission time is long and the system, during its mission time should not only deal with ordinary situations but also must be able to recover from some exceptional situations.

In view of the above characteristics, the design of real-time systems poses serious challenges. There is a definite need for systematic methods and methodologies for designing them. In the design of quality software, high level programming languages and abstract models have a major role to play. The focus of this monograph is on some of the high level programming abstractions that have been found to be useful in designing provably correct real-time programs.

1.1 Real-time and Reactive Programs

There are many dichotomies of programs such as determinism/non determinism, synchrony/asynchrony, off-line/on-line, virtual time/ real-time, sequential/concurrent. However, depending on the way they interact with their environment, programs can be classified into the following three broad kinds[9]:

1. *Transformational Programs*: These programs compute outputs given the input; programs interact with their environment once at the beginning to get inputs and once at the end to give outputs. Compilers are examples belonging to this category.

2. *Interactive Programs*: These are programs that interact at their own speed with users or with other programs; time-sharing operating systems are typical examples of this category.
3. *Reactive Programs* : These programs maintain a continuous interaction with their environment, but at a speed which is determined by the environment (and not by the programs). In other words, the output may affect future inputs due to the *feedback* inherent with continual interactions. Real-time programs are sub-classes of reactive programs wherein hard-clock sensor values are needed and the correctness of the programs depends on the speed of the sensors and processors.

Traditional programs like data processing application programs and numerical computations are transformational in nature, as they describe transformations of values of variables (in discrete steps). Any processor implementing these transformations takes a nonzero finite amount of time. These programs, by their very definition, are time independent and hence, designed such that the computed results are independent of the execution speed of their processor(s). In other words, time considerations are completely irrelevant for the functional behaviour of these programs and their correctness; it is only relevant for questions of efficiency.

A typical example of an interactive program is a time-sharing operating system. An operating system consists of a number of sequential programs called processes that run concurrently and interact with each other. Some of these processes interact with I/O devices while some others execute user programs. The rate at which different processes are executed depend, besides processor speeds, upon the process scheduler. A process involving a lot of I/O will be slow compared to a process which does pure computation. Also, the processes executed at different rates may need to interact; for instance, a user program sending a sequence of inputs to another process that stores these inputs onto a disk. A naive design of such systems would take into account explicitly the speeds of different processes. But such a design is not desirable as it will not be robust to the changes of the processors or the schedulers or even a change in a specific algorithm being used in a process. Consequently, the design of interactive systems has always proceeded by abstracting the execution speed (or time) of processes with the result that correctness of programs is not dependent upon the processor speeds and hence time; of course, a price to be paid in such an abstraction is that programs exhibit nondeterministic behavior.

A classic example of a reactive system is the quartz digital watch. In a digital watch, quartz vibrations and button pressures (i.e., the environment)

determine the watch behaviour; the environment plays a vital role in reactive programs as compared to interactive programs. As an aside, it is important to note that interactive programs have reactive components. For example, the terminal driver in an operating system is a reactive program.

For real-time systems, it is not necessary to use clock-times. For example, consider the description of a braking system of a train. One could say

when brakes are applied the train should come to halt within 200 meters.

In fact, the above specification is more acceptable than the following clock-time based specification:

when brakes are applied the train should come to halt within 2 sec.

In the latter specification, one has to verify whether the specification is meaningful in the context of trains with different speeds.

The most important reason for considering time explicitly is in modelling some physical processes wherein the internal laws that define the *natural* behaviour of the physical process are functions of *physical time*. It is in this context one enters the field of real-time programming. In recent times, *hybrid* systems which are a combination of discrete and continuous systems have become important. In these systems, the reference to global clock-time is again important. Wirth showed in [90], that there is a need to consider explicit time/speed even in some concurrent programming situations wherein all the logical processes cannot be physically mapped onto processors due to some hidden assumptions on synchronization.

From the design point of view, transformational and interactive programs have similar characteristics and their behaviour can be described by mathematical functions. Reactive systems are quite different from these two classes. In real-time systems, timely execution of requests and responses by the controllers is critical to the successful operation of both the physical systems and the computer itself. That is, in addition to the normal functional requirements, it is necessary that any response to an input (from the environment) must happen in a given interval of time, called its *deadline*.

Deadlines are classified into hard- and soft-deadlines. A deadline in a program or a system is said to be *hard* if it is mandatory for the program/system to meet the deadlines. In other words, violating the deadlines leads to failure. A deadline is said to be *soft* if missing the deadline does not compromise the correctness of the program but possibly degrades the

performance of the system. Systems with hard (or soft) deadlines are called hard (or soft) real time systems . Our emphasis in this monograph is on the development of correct hard real time systems.

The need for robustness and fail-safeness calls for a sound and systematic methodology for the design of reactive programs. In fact, one of the main hurdles in the development of a sound methodology is that the well-known principle of *separation of concerns*, is difficult to follow. For example, even a small real-time system, say, a tactical embedded system for an aircraft might be simultaneously maintaining a radar display, calculating weapon trajectories, performing navigation functions etc. In these systems, one sees that:

1. the software implementing the various tasks are mixed together and it is difficult to determine which task is performed by a part of the code
2. the timing dependencies between tasks are such that changing the timing characteristics of one task may affect other unrelated tasks meeting their deadlines.

In real-time systems, the functional and timing requirements are interwoven. In view of this, the classical approaches of software development cannot be relied upon. We suggest the use of rigorous formal methods for the various phases of software development ranging from specification to realization. The monograph addresses some of these aspects.