Computer Science 773

Robotics and Real-time Control

WHAT IS REAL-TIME ?

REAL-TIME AS OPPOSED TO WHAT ?

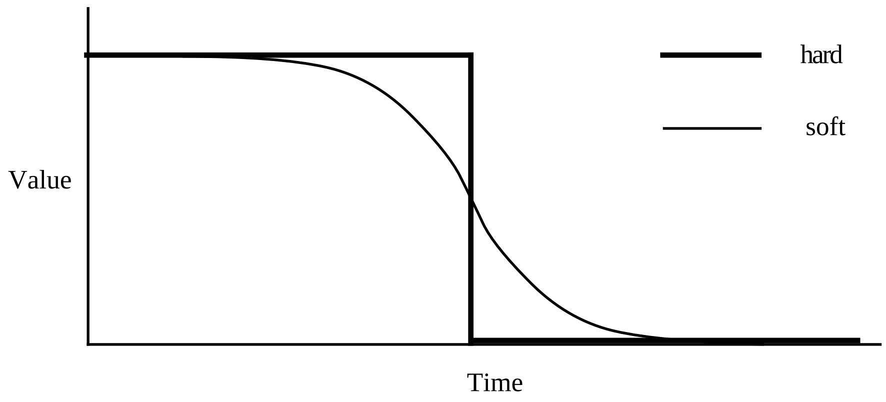
We don't normally speak of unreal-time or imaginary-time programming, but if we did we'd mean programming in which time was not of direct importance. While we would, presumably, still prefer our programmes to finish in as short a time as possible, nothing will necessarily go wrong if they don't.

In a real-time programme, time is important : there is some sort of deadline by which we expect that some task will be complete. It might or might not cause irreparable damage if the deadline is not met, but something will go wrong somewhere. The deadline is usually (always ?) imposed by some process external to the computer which is not under the programme's complete control, and with which the programme must synchronise somehow.

VARIETIES OF REAL-TIME.

Real-time comes in two flavours : hard and soft. In fact, a less dramatic but more precise statement would introduce a hard-soft continuum, and imagine each real-time programme as lying somewhere along the line.

The "hardness" of a real-time programme is a measure of the urgency of its deadlines. Imagine a graph showing the value of some result against completion time : then the hardness of the programme is related to the maximum steepness of the curve. Hard real-time programmes typically have to do with controlling machines, while soft real-time is to do with people – as in network systems dealing with terminal transactions.



One can speculate that the hardness is a consequence of the stupidity of the other (non-computer) party to the transaction. When the other party is human, and therefore comparatively intelligent, delays can be recognised, interpreted, and sensibly handled, but machinery is usually stupid, and less adaptable. In a large, perhaps hierarchic, system, different degrees of hardness might be found in different parts; typically, the hardness decreases as the distance from the machinery increases.

CHARACTERISTICS OF REAL-TIME.

Some differences in emphasis between hard and soft real-time computing :

	HARD	SOFT
Critical requirement :	Keep up with events	Provide good service
Problems :	timing	communications
	priorities	distributed databases
	resource allocation	integrity
	scheduling	
Emergency priorities :	regain control	ensure integrity

REAL-TIME PROGRAMMING COMPARED TO OTHER SORTS.

Niklaus Wirth suggested that there was a hierarchy of complication in various applications of programming. He distinguished three cases :

Ordinary sequential programming : Problems can be solved, or they can't. If they can, complexity theory tells us something about the efficiency. We've developed lots of useful knowledge about how to write programmes, and we want to keep using it.

Concurrent programming : Ordinary programming + synchronisation. There are no (serious) problems if different threads don't interact. If they do, there are possibilities both of conflict and of faster completion – but, apart from deadlock, in the worst case we are usually no worse off than with sequential programming. With concurrent systems, we would like to keep using the sequential programming skills, and to add the synchronising parts as a separate package. We want :

Sequential computing + synchronisation = concurrent programming.

Can we have it ? We can certainly get quite close, provided that we interpret "synchronisation" fairly liberally. (For example, it has to include mutual exclusion.).

Real-time programming : Concurrent programming + timing. The rate of execution of the code is no longer determined only by the properties of code and computer, but also by interactions with entities which act independently of the computer. Now we can't avoid problems, because there is a new sort of constraint : you might not have time to do the calculations. The worst case is failure. Nevertheless, it is still sensible to try to build on what we can do. Can we preserve both sequential and concurrent programming skills and separate out the timing ? This time, we want :

Concurrent programming + time constraints = real-time programming.

Wirth argued that the same method could be used again; real-time computing can be reduced to comparatively time-insensitive concurrent programming by providing further primitives which encapsulate the awkward details of critical time dependencies. He put these ideas into practice in designing the languages Modula and (later) Modula-2, which were specifically intended for real-time systems, and were very effective.

But the solution wasn't perfect. In fact, the two steps of increasing complexity differ significantly in a way which Wirth's argument does not take into account. In a multiprogrammed system, there are several processes which impose constraints on each other through requirements of the form "Process A cannot proceed past point P until process B has reached point Q". Except for these purely logical constraints, the proceses do not interact; in particular, the different processes proceed independently, and there is nothing in the constraints which will prevent the required conditions from eventually being met. (I assume that the system concerned is in principle feasible, so that deadlock can be ignored.) In a real-time system, this is not so. The requirements now are of the form "Process A must reach point P before time T", but these requirements are no longer independent for processes which share a processor, as the more time we allocate to process A to satisfy its requirement, the less we can give to process B. There is no way to avoid this direct interaction between the processes (short of providing more processors, a solution not within the ambit of software engineering), and no amount of clever software will take it away.

For the time being, then, we must adopt an empirical approach. Wirth's analysis at least points out the importance of time in real-time computing, and in practice this turns out to be a prominent feature. The other main novelty is the requirement that real-time computers must communicate with the systems they control, so problems of interfacing become significant. These fundamental questions lead on to related topics; the following notes review these briefly.

FIRST-ORDER SPECIAL THINGS.

Event-driven computing. Much of real-time computing is driven by events, in the sense that programmes must wait for some condition to be satisfied before starting. The conditions may depend on explicit signals received as interrupts from the controlled system, or on values read from the controlled system, or on internal variables maintained by the computer. The events constrain the times at which actions can start.

Time limits. A process once started typically must be completed within some specified time. These limits are the duals of the events, in that they constrain the times by which actions must stop.

Communications. A real-time system must usually exchange information with the object which it controls. This may require interfacing with any form of signal, particularly with hard real-time systems which deal directly with machinery of one sort or another. At higher levels in a system, as the timing requirements become less stringent, standard protocols may be used, as most communications will be between computers.

SECOND-ORDER SPECIAL THINGS.

Under this heading, we can consider responses to the characteristics of real-time computing listed above. We might expect to find programming disciplines and software engineering techniques appropriate to the topics mentioned. There are indeed some such methods, though they are not as well understood as the corresponding topics in conventional computing.

Apart from fundamental problems of the sort discussed in connection with Wirth's work, there are practical difficulties. Many real-time programmes run on dedicated microprocessors embedded in machinery of various sorts, and without facilities for systematic programme development and debugging. It is therefore not uncommon for programmes to be developed in larger machines, and to be transferred to their proper environments only when considered fairly complete. Proper testing is difficult under these circumstances, and fault correction may involve a cumbersome cycle of testing on the real system, followed by diagnosis and correction on the development machine, after which a new version for the real system can be produced. System emulators on the development machine can help, but it remains difficult accurately to reproduce the production environment for thorough off-line testing.

REFERENCE.

N. Wirth : "Toward a discipline of real-time programming", *Comm. ACM* **20**, 577 (1977).

Alan Creak,
March, 1998.