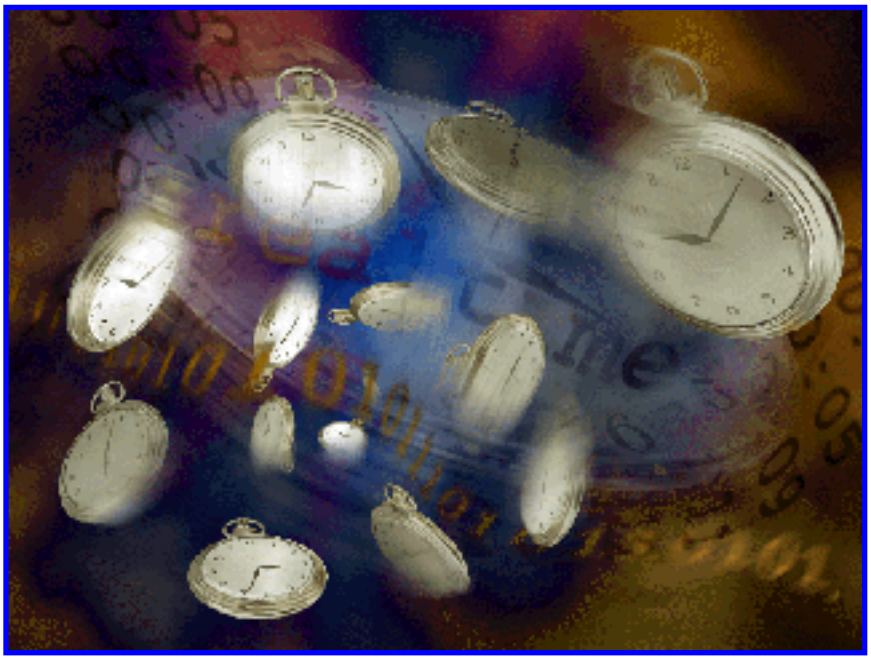
It's Now or Never

by C. Fidge



`Real-time computing: a new discipline of computer science and engineering.' With this title a recent technical journal article [5] neatly summed up current research on real-time program development. The field is not only new, but involves a challenging combination of traditional computer science and engineering methods.

So what is real-time computing? In short, any situation where the time at which a result is produced is as important as the result itself. Producing the right answer too late is as bad as producing the wrong answer or no answer at all! Avionics, robotics and process control are all examples of real-time computing applications.

Of course, `time' has been an issue in computer science since the days of punched cards and paper tape. Everyone has always wanted their program to run `fast'. But general performance goals are *soft* timing requirements only; a missed deadline is usually not serious. If an airline reservation system takes a few seconds longer than average to make a booking the only consequence may be a disgruntled passenger.

Hard real-time requirements, on the other hand, demand not only efficiency but the ability to predict how long each computation will take *in advance*; in this case a missed deadline may be life threatening. If a microprocessor-controlled heart pacemaker takes a few seconds longer than expected to initiate treatment the consequences can be disastrous.

This leads us to the major precept of real-time computing. Predictability is more important than performance!

This is often overlooked. For example, modern computer processors intended for real-time applications aim to run as fast as possible. To do this they incorporate performance-enhancing hardware features such as instruction pipelines and memory caches. Unfortunately, although they improve average speed, these devices mean particular assembler instructions may have different timing behavior in different situations. Ironically this makes it very difficult to accurately predict the timing behavior of programs running on such machines.

At present real-time software is difficult, and thus expensive, to produce. Programmers are forced to work with low-level machine languages because conventional programming aids hinder, rather than help, the ability to guarantee timing predictability. An optimizing compiler, for instance, may rearrange the object code it generates to improve performance. However this makes it difficult for the programmer to accurately predict the timing behavior of a program from its source code. Also real-time systems must undergo an unusually long testing phase to increase confidence in their timing behavior. Yet even this is not always sufficient because the system may not have exactly the same real-time behavior under test conditions as it does `in the field'.

At the Software Verification Research Centre we are solving these problems by developing formal software engineering methods for systematically producing verified concurrent real-time software. Our *Quartz* and *Topaz* projects are using verification techniques to achieve timing predictability, and rigorous development rules for efficient production of real-time software. Established real-time software engineering principles such as schedulability testing are incorporated into the procedure. The difficulties posed by hard-to-predict hardware timing are approached by creating accurate timing models of the target processor.

Formal development methods for untimed, sequential computer programs are well-understood. They allow a formal requirements *specification* to be systematically transformed into a program code *implementation* by application of semantics-preserving *refinement* rules. Each such rule carries a, frequently implicit, proof obligation. Thus program development and verification proceed in lockstep. `Wrong turns' in program development are detected early. Refinement is thus far more efficient than fully developing a system and then undertaking a large and complex post-verification effort.

To date such rules have not been suitable for real-time or multi-tasking applications, however. Therefore we are building on these methods by introducing time as a limited computing resource like any other.

The Quartz method allows an abstract real-time specification to be refined to high-level language program code. The specification expresses the desired behavior of variables as traces, or histories, with absolute time as the trace index. A rigorous development procedure then allows this specification to be transformed into a skeletal multi-tasking program, annotated with precise timing constraints that each code segment must obey. Program development is not considered complete until it has been formally proven that each code segment executes within these constraints.

The Topaz method will take care of this by treating program compilation as a formal `refinement'. Here the time-annotated high-level language program is our `specification' and the executable assembler code is our target implementation. As each time-constrained statement is translated into assembler code the timing requirements associated with it will be checked against a real-time model of the target machine. If all the constraints are satisfied we know, without even running the program, that our original real-time requirements will be met!

In summary then, real-time computing *is* a challenge, but not an insurmountable one. However, while we can often enhance the functional behavior of a computer, by adding more memory or a new arithmetic unit, we cannot alter the passage of time. Our development methods must therefore adjust our software to accommodate this `resource', rather than vice versa. Time remains as much an immutable force in computer science as in our everyday lives.

References

1

Fidge, C., Kearney, P., and Utting, M., Interactively Verifying a Simple Real-time Scheduler, in *Computer-Aided Verification: 7th International Conference*, Springer-Verlag Lecture Notes in Computer Science 939, pp. 395-408, 1995.

2

Fidge, C., Utting, M., Kearney, P., and Hayes, I., Integrating Real-Time Scheduling Theory and Program Refinement, *Formal Methods Europe '96*, Springer-Verlag Lecture Notes in Computer Science, 1996. (To appear.)

3

IEEE Software, Special Issue on Real-Time Systems, Vol. 9, No. 5, September 1992.

4

Morgan, C., *Programming from Specifications*, Prentice-Hall, 1990.

5

Shin, K., and Ramanathan, P., Real-time Computing: A New Discipline of Computer Science and Engineering, *Proceedings of the IEEE*, Vol. 82, No. 1, pp. 6-24, January 1994.

Colin Fidge (cjf@cs.uq.oz.au) is a Senior Research Fellow with the Software Verification Research Centre, Department of Computer Science, University of Queensland, Australia. His research interests include formal methods, real-time systems, and models of causality in distributed systems.

The Quartz project is funded by the Defence Science and Technology Organisation.