PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

Some directions in concurrency theory (Statement for panel on "Theory and Practice in Concurrency").

Robin Milner

Department of Computer Science, Edinburgh University
King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, U.K.

I want to comment on two aspects in which a theory of concurrent computing is different from a theory of sequential computing. In both aspects, we see the same difference: namely, that the former is about the whole of computation, while the latter is only about a slice of it.

The first aspect is to do with the computer-in-context, as opposed to the computer by itself. There used to be, and largely still is, a strong division among three types of description. First, a sequential program was a high-level description of what can occur in a computer loaded in a certain way. (If you don't like calling a program a 'description' then consider how else, other than just by presenting the program, you could describe the possible sequences of memory transfers that a sequential program can perform.) Second, automata theory or engineers' diagrams described what can go on in the same machine at a *lower level*, and this certainly isn't sequential. Third, narrative prose – or any scientific notation suitable to the applications – described what may occur *outside* the same machine between, or during, program executions; this, too, is only rarely sequential. It took some time – thanks to von Newmann's bottleneck (which, however, got computing off the ground) – before we thought that the same descriptive medium might serve all three purposes. Carl Petri is primarily responsible, via Net Theory, for giving us hope for a unified theory in which to describe and analyse *all* aspects of the computer-in-context (computer + program + aircraft, or computer + program + banking staff) rather than just the computer itself, or just the program itself. This has to be a theory of concurrency, because of the three ingredients *at most* one – the program – is sequential!

The subject matter for such a theory is vast – all discrete dynamic systems – and it is natural to expect specialised theories, with special notations, for particular classes of system. Not all these theories are the concern of the computer scientist; but there is a distinct challenge for computer scientists – namely that they must contribute on two levels. At the lower level they must build the specialised theory for those subsystems which happen to be computers or programs. At the upper level they must provide the global theory, the general theory of concurrency into which all the specialist theories must fit; there must be a theoretical framework which can embrace special theory of information-flow among banking staff, the special theory of programs, and all the other special theories. But it would be untidy and unfortunate if these two levels were fundamentally different! Since a significant part of the general theory must be a tractable descriptive notation, and since we have already classified programs as descriptions, one hopes that concurrent programming languages will be nothing more than a part of the descriptive machinery of a general theory of concurrency. My point is that the barrier to this unification is removed as soon as programs are not forced to be sequential, and we must exploit this freedom.

The second aspect in which concurrency is about the whole of computation is in its concern with structure. A proper theory of concurrency must explain the structure (division into processes) of a program, as well as the structure (division into processors) of a computer, and the relationship between the two. Sometimes – perhaps in solving partial differential equations, or in some large physics calculations, or in weather-forecasting programs – the process structure of the programs can be fixed and simple (e.g. grid-like) and there can be a fixed allocation of processes to processors. Other applications – much more interesting to a computer scientist – are not like that; for example, an operating system program together with all the programs it runs, or the description of architectures such as the ALICE machine which aim at parallel execution of declarative programs. For these applications, if we wish to analyse them thoroughly, we have to find tractable descriptive methods and in which both the virtual (program) and the real (machine) processes are written in the same terms, and in which two kinds of mobility can be reflected: the changing population and linkage of the virtual processes, and their shifting allocation to the processors. It seems only with concurrency that we arrive at, and wish to tackle, this very subtle problem of relating two distinct ways of structuring the behaviour of a complex system.

It is true that some people – including Hewitt, and Kennaway and Sleep – have given notations in which a significant amount of this mobility can be written down. The big challenge, though, remains to find the right mathematics in which to analyse these descriptions; I think that this needs great innovation.

In this second aspect, just as in the first, we can take advantage of the idea that programs are descriptions. When we have found a good mathematical means to describe the mobile structure among processes and their changing association with processors, we shall almost certainly find that we can enrich our concurrent programming languages by absorbing the new descriptive notations into them. It is in this way that programming has become richer in the past, by absorbing the descriptive notations of logical and of functional processes, and I see no reason why it should not happen, even more excitingly, with the notations of various aspects of concurrent processes. I therefore see concurrency theory as a means of coming to understand, through structure, processes (both built by us and naturally existing) which have previously been beyond our grasp.