disc; further, the 'split cylinder' technique can be used to reduce time-consuming seeks in this case. For example, a two-disc configuration might match the throughput of a five-tape configuration (but possibly might still be the more expensive of the two).

3. Even sequential disc files have the flexibility of direct access when required, if only by the 'binary chop' technique; this facility might save passes of the file. For example, amendments to record keys can require that the file be resequenced in the current run; a tape file will require an extra pass to achieve this, whereas the direct access facility can be used to avoid this for the disc version.
4. There are many more arguments that favour disc sequential processing but, on the other hand, tape sequential processing often (but not always) proves to be more economical. Generally, a costed saving in computer run time must be weighed against increased hardware and stationery costs.

Conclusion

The rules of thumb have been disproved. A methodology of computer systems design must necessarily commence with careful consideration of the objectives and continue through an iterative, decision-making procedure which attempts to meet these objectives; the CAM project is developing such a methodology. It appears doubtful that any short-cuts can justifiably be taken to achieve an efficient design other than stopping the design procedure when an acceptable solution has been achieved.

Finally, the author wishes to acknowledge the assistance of his colleagues at LSE, particularly Mr. F. F. Land and Dr. A. H. Land of the Statistics, Mathematics, Computing and Operational Research Department. The author would be pleased to discuss the details of this paper with any interested organisation, particularly in relation to specific computer systems design problems.

References

DANIELS, A., and YEATES, D. (1969). *Basic Training in Systems Analysis*, London: Pitman (for National Computing Centre).
LOSTY, P. A. (1969). *The Effective Use of Computers in Business*, London: Cassell.
MARTIN, J. (1967). *Design of Real-Time Computer Systems*, New Jersey: Prentice-Hall.
WATERS, S. J. (1970). Physical Data Structūre. Paper 6 of Proceedings of BCS Conference on Data Organisation.
WATERS, S. J. (1971). Blocking Sequentially Processed Magnetic Files, *The Computer Journal*, Vol. 14, pp. 109-112.

Correspondence

To the Editor
The Computer Journal

Sir,

High level languages are unnecessarily complex for the inexperienced user, D. G. Evershed and G. E. Rippon say in a paper in Vol. 14, No. 1, 1971 of this *Journal*. The authors of the paper discuss a number of modifications to FORTRAN, ALGOL and other high level languages. The goal of most of the suggested modifications is to make it easier to write programs which are acceptable to the compiler. Many of the suggestions made by the authors are very valuable. However, the authors do not seem to be aware of a basic conflict underlying some of their suggestions.

The basic conflict is the following: The errors in a computer program can be divided into two categories:

1. Language errors, which can be detected by the compiler.
2. Logical errors, which can only be detected by erroneous results during the execution.

The second category, the logical errors, are much more troublesome than the language errors. The reason for this is that the compiler gives a good and useful diagnostic for most language errors, which makes it very simple for the programmer to correct his errors. For logical errors, on the other hand, the process of finding and correcting them is often much harder. There is even a large risk that logical errors are not discovered until production use of the program has begun. Sometimes, logical errors are not discovered at all, which means that the results of the runs of the program may be dangerously false.

Because of this, it is much more important to design a computer system which gives few logical errors than to design a system which gives few language errors.

In many cases, a programming language construct can be designed either to give few language errors or to give few logical errors. The reason for this is that the language can be designed in such a way that as many common logical errors as possible will also cause language errors. If a programming language is designed in this way, then the compiler has much larger possibilities to help the programmer avoid logical errors.

A very simple example; the following ALGOL program contains an error:

```
integer abcde;
.....
abcde := abode + 1;
```

The intention of the programmer was to increase *abcde* by 1. However, by mistake, he instead wrote *abode* on the right hand side. This may cause a logical error which is very difficult to discover, if the sequence above was written in FORTRAN. However, in ALGOL, a language error occurs: *abode* is an undeclared variable. Thus, the compiler can detect the programming error in ALGOL but not in FORTRAN.

A second example; the following construct is allowed in both ALGOL 60 and FORTRAN but not in ALGOL 68:

```
real a; integer i;
.....
i := a;
```

The difficulty with this statement is that real variables can be converted into integers in many different ways. You can make a correct rounding (using different ways of rounding) or you can take the nearest lower integer, either with sign (as the ALGOL *entier* function) or without sign (as the FORTRAN *int* function). In fact, the program above will be executed in one of these ways in ALGOL 60 and in another way in FORTRAN. A very common programming error is to assume one conversion when the real conversion is another than the one assumed. This error cannot occur in ALGOL 68.

A third, more complex example, is the handling of pointer variables in various languages. If these pointer variables are typedeclared and typechecked (like in ALGOL 68 and in Simula 67) then the risk of undetected logical errors is much smaller than without such checking (like in PL/1 or in SIMSCRIPT). Also, a garbage collector (like in LISP, SNOBOL, ALGOL 68, SIMULA 67) gives smaller risk of programming errors than explicit deallocation of list structure records (which is done in PL/1 and SIMSCRIPT).

(Continued on page 36)

Downloaded from https://academic.oup.com/comjnl/article/15/1/4/418391 by guest on 19 April 2024

time slicing strategy is further modified to include a 'least time to go' strategy for the end of the interval, then a tolerance $\tau^{(1)}$ will be produced for that interval. Condition 1 implies that if $\tau^{(1)} < \tau$ then a quantity $(\theta^{(1)}/\Theta)\,\tau^{(1)}$ of processor activity at the end of $\Delta T_i^{(1)}$ is freed for utilisation by process $P^{(2)}$. It is easily shown that a tolerance $\tau^{(2)}$ is allowed in the time to spare before the next event in $P^{(2)}$ if

$$\tau^{(2)} \leqslant \min\left(\tau, \frac{\theta^{(1)}}{\theta^{(2)}}\tau^{(1)} + (1 - \Theta)\,\Delta T_i^{(2)}\right)$$

This may be summarised by
Condition 3:

$$\exists \tau^{(1)}, \tau^{(2)} \leqslant \tau$$

such that

$$\tau^{(2)} \leqslant \frac{\theta^{(1)}}{\theta^{(2)}}\tau^{(1)} + (1 - \Theta)\min_i \Delta T_i^{(2)}$$

$$\tau^{(1)} \leqslant \frac{\theta^{(2)}}{\theta^{(1)}}\tau^{(2)} + (1 - \Theta)\min_i \Delta T_i^{(1)}$$

A more compact but stronger condition is
Condition 3*a*:

$$\tau(1 - \min\left(\frac{\theta^{(1)}}{\theta^{(2)}}, \frac{\theta^{(2)}}{\theta^{(1)}}\right) \leqslant (1 - \Theta)\min_i \Delta T_i)$$

Conditions 1, 2 and 3a are adequate to show that the set-up in which $\theta^{(1)} = \theta^{(2)} = \frac{1}{2}$ and $\Delta T^{(1)}{:}\Delta T^{(2)} = 3{:}2$ with events maximally staggered leads to a tolerance of one sixth of the longer period.

More than two processes

With a greater number of processes ($n > 2$) much more complex combinations of possibilities exist. The problem may be regarded as involving combinations of $n(n - 1)/2$ pairs of processes, each pair using a share $2/n(n - 1)$ of the total processor power. The argument implied is again based on systematic development of a strategy starting with 'infinitesimal time slicing'. The tolerance τ is at least that time in any interval which can be saved by some pair of processes out of its limited power $2/n(n - 1)$. A weak but easily derived set of conditions is therefore as follows:

Condition 1*:

$$\tau \leqslant \frac{2}{n(n-1)} \max_{j'} \left|T_i^{(j)} - T_{i'}^{(j')}\right| \text{ all } j, i, i'$$

Condition 2*:

$$\tau \leqslant \frac{2}{n(n-1)}(1 - \theta^{(j)})\,\Delta T_i^{(j)} \quad T_{i'}^{(j')} < T_i^{(j)} < T_{i+1}^{(j)} < T_{i'+1}^{(j')}$$

Condition 3*a**: $\exists j, j'$
such that

$$\tau\left(1 - \min\left(\frac{\theta^{(j)}}{\theta^{(j')}}, \frac{\theta^{(j')}}{\theta^{(j)}}\right)\right) \leqslant \frac{2}{n(n-1)}(1 - \Theta)\min_i \Delta T_i$$

References

BCS Specialist Group (1967). A language for real-time systems, *The Computer Bulletin*, Vol. 11, No. 3, pp. 202-212.
Conway, R. W., Maxwell, W. L., and Miller, L. W. (1967). *Theory of scheduling*, Addison-Wesley Publishing Company.
Fineberg, M. S., and Serlin, O. (1967). Multiprogramming for Hybrid computation, Proc. AFIPS, Fall Joint Computer Conference.
Jackson, J. R. (January, 1955). Scheduling a production line to minimise maximum tardiness, Research Report 43, Management Sciences Research Project, U.C.L.A.
Middleton, M. D. (March, 1971). A Note on Scheduling Real Time Processes, Cambridge University Engineering Department.
Wirth, N., and Hoare, C. A. R. (1966). A contribution to the development of Algol, *CACM*, Vol. 9, No. 6, pp. 413-31.

Correspondence

(Continued from page 4)

The basic principles behind these examples are two:

1. A programming language should include much *redundancy*, which means that when a correct program is erroneously modified, then as many of the logically false modifications as possible should introduce not only logical errors but also language errors into the program.
2. When a programming language construct is *ambiguous*, that is can be interpreted in more than one way (like *a* := *i* in the example above), then the whole construct should be forbidden, forcing the programmer to use other, unambiguous constructs instead (like *a* := **round** *i* in ALGOL 68).

These two principles have one common aspect: The larger ability to detect errors for the compiler is gained by restricting the freedom of the programmer to write anything he likes and have the compiler try to understand what he means. Such an *understanding* compiler is very dangerous, since it will not be as good at detecting logical errors.

This is the basic conflict which the authors of the paper in your February issue did not discuss. The authors want to change high level languages to be more *understanding*. But the disadvantages with such changes are sometimes much larger than their advantages. Especially with the advent of time-shared computers, the correction of compiler-detected errors becomes very simple. The real risk is not the compiler-detected errors, as the authors seem to think, but the logical errors not discovered by the compiler.

Yours faithfully,
J. Palme

Datalogy Section
Research Institute of National Defense
S-10450 Stockholm 80
Sweden
16 August, 1971

References

Palme, J. (1969). What is a good programming language? FOA P Report C 8231. Research Institute of National Defense, S-10450 Stockholm 80, Sweden.
Palme, J. (1971). Simula 67—An advanced programming and simulation language, Norwegian Computing Center, Forskningsveien 1b, Oslo, Norway.

Downloaded from https://academic.oup.com/comjnl/article/15/1/4/418391 by guest on 19 April 2024