A sequel to EWD535.

I just realized that in EWD535 I have failed to touch upon the second point raised in your letter, where you write that you feel compelled to

replace $\quad$ **if** B → S1 [] **non** B → S2 **fi** $\quad$ (1)

by $\quad$ **if** B **then** S1 **else** S2 **fi** $\quad$ (2)

because of the fact that two successive calls of B "may not return the same value on two calls because of another precessor changing the state". I had forgotten about it, because in the EWD535-versions of your program the boolean procedure had disappeared.

In my book "A Discipline of Programming", and in all the theory about semantics that underlies the notion of the guarded commands, all expressions --not only the guards, but also all arithmetic expressions at the right-hand side of an assigment operator-- are regarded as (possibly partial) functions of the "current state", which is supposed to change only as a result of explicit assignments. Furthermore I have restricted myself to a programming language that trivially admits a sequential implementation. I have given no further prescriptions about that implementation; in particular I have not prescribed that the execution of (1) must imply a separate evaluation of B and another one of **non** B . On the contrary! One can defend the point of view, that for any boolean expression B the evaluation of B by definition implies the concurrent evaluation of **non** B , because both answers give exactly the same information about the current state. From that point of view --I still talk about sequential uniprogramming-- one can appreciate (2) as a hint to that part of the compiler that optimizes boolean expressions: it saves it the trouble to recognize that the two guards are the complement of each other. Needless to say, that all forms of side-effects are ruled out: they are regarded by me as invalid implementations as they would violate the axiom of assignment, etc.

The awkward point is how to transfer this pattern of reasoning in order to describe the semantics of a number of mutually unsynchronized programs that --at a certain grain of interleaving, say: a memory cycle-- fool in the same store. I know of only one way (and it is not very attractive! I shall sketch it nevertheless; the unattractiveness is probably a consequence of the sad fact that these problems are inherently ugly.)

Consider programs A and B with the shared variables x and y; consider then separately program A in its private state space extended with x and y , and the program B in its private state space extended by x and y. When considering program A we now must admit that at each semicolon of A so to speak, the total state of A (i.e. including x and y) may change non-deterministically, only bound by the limitations of what B may do. If, for instance, B has the trivial form

$$\underline{\text{do}}\ \text{true} \rightarrow x := \text{random}\ \underline{\text{od}}$$

(assuming that B has a private random number generator) this means for program A that at any semicolon the value of x may be subjected to a random variation. (A rather terrible form of interference!) If B has the trivial form

$$\underline{\text{do}}\ \text{true} \rightarrow x := 1\ \underline{\text{od}} \qquad (3)$$

it means for program A that at any "semicolon" x is either equal to 1 or unchanged. In order to make this a workable system, one has to postulate, that in each "unit of evaluation" (see below) at most one shared variable is referenced at most once. Here a unit of evaluation is something about the internal sequencing of evaluation we don't wish to make any committments. Without that constraint it could make a difference whether A evaluated "x + x" or "2 * x": in the second case we could guarantee an even result, in the first case we can not! And then, a more or less decent mathematical system becomes totally impossible.

The problem, of course, is, that program B has more structure than (3), and that, when studying program A we have to take that into account. If, in program A we have

$$\underline{\text{do}}\ x > 0 \rightarrow s := x;\ x := s - 1\ \underline{\text{od}} \qquad (4)$$

we cannot guarantee termination with B of form (3) ; we could however, with B of the form

$$\underline{\text{do}}\ \text{true} \rightarrow \underline{\text{do}}\ y > 0 \rightarrow x := 1\ \underline{\text{od}}\ \underline{\text{od}} \qquad (5)$$

guarantee termination of (4) provided initially $y \leq 0$, because then B can interfere with the loop (4) at most once. More detail you can find in the thesis of Susan Speer Owicki from Cornell University, Ithaca, N.Y. 14850 (according to my Webster), Department of Ccmputer Science.

It was the experience cf studying her thesis, and the moral of a number of my own exercises, that caused me --first thing I did!-- when I

tried to understand your solution, to do away with the boolean procedure INITIALIZERESPONSIBILITY: the value that its call returns is not a function of the state, but a (very complicated) function of past history. But that implies that I want to see at this level the semicolons, the sequencing to be more precise. One can read your observation as a plea for the if-then-else-fi construct; another conclusion can be that a function procedure, the evaluation of which references more than one shared variable is a misleading construct that we had better regard as "against the rules". Susan Owicki has made the latter choice, and she has my blessing. Unless new arguments emerge I think that I shall stick to my guarded commands: I am still quite happy with them!

The most effective way of mastering complexity is avoiding the introduction of complexity in the first place! I would love to know how I could put more "meat" into that observation.

Yours ever,

Burroughs
Plataanstraat 5
NL-4565 NUENEN
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow