Acta Informatica 1, 115–138 (1971)
© by Springer-Verlag 1971

Hierarchical Ordering of Sequential Processes

E. W. Dijkstra

Summary. One of the primary functions of an operating system is to rebuild a machine that must be regarded as non-deterministic (on account of cycle stealing and interrupts) into a more or less deterministic automaton. Taming the degree of indeterminacy in steps will lead to a layered operating system. A bottom layer will be discussed and so will the adequacy of the interface it presents. An analysis of the requirements of the correctness proofs will give us an insight into the logical issues at hand. A "director-secretary" relationship will be introduced to reflect a possible discipline in the use of sequencing primitives.

0.

The processing unit of a working computer performs in a short period of time a sequence of millions of instructions and as far as the processing unit is concerned this sequence is extremely monotonous: it just performs instructions one after the other. And if we dare to interpret the output, if we dare to regard the whole happening as "meaningful", we do so because we have mentally grouped sequences of instructions in such a way that we can distinguish a structure in the whole happening. Similar considerations apply to the store: high speed stores contain typically millions of bits stored in a monotonous sequence of consecutively numbered but otherwise equivalent storage locations. And again, if we dare to attach a meaning to such a vast amount of bits, we can only do so by grouping them in such a way that we can distinguish some sort of structure in the vast amount of information. In both cases the structure is *our* invention and *not* an inherent property of the equipment: with respect to the structure mentioned the equipment itself is absolutely neutral. It might even be argued that this "neutrality" is vital for its flexibility. On the other hand, it then follows that it is the programmer's obligation to structure "what is happening where" in a useful way. It is with this obligation that we shall concern ourselves. And it is in view of this obligation that we intend to start with a rather machine-bound, historical introduction: this gives us the unordered environment in which we have to create order, to invent structure adequate for our purposes.

In the very old days, machines were strictly sequential, they were controlled by what was called "a program" but could be called very adequately "a sequential program". Characteristic for such machines is that when the same program is executed twice—with the same input data, if any—both times the same sequence of actions will be evoked. In particular: transport of information to or from peripherals was performed as a program-controlled activity of the central processor.

With the advent of higher electronic speeds the discrepancy in speed between the central processor on the one hand and the peripheral devices on the other became more pronounced. As a result there came for instance a strong economic pressure to arrange matters in such a way that two or more peripherals could be running simultaneously.

In the old arrangement one could write a program reading information from a paper tape, say at a maximum speed of 50 char/sec. In that case the progress through that piece of grogram would be synchronized with the actual movement of the paper tape through the reader. Similarly one could write a program punching a paper tape, say at a maximum speed of 30 char/sec. To have *both* peripherals running simultaneously and also closely to their maximum speed would require a tricky piece of program specifically designed for this mix of activities. This was clearly too unattractive and other technical solutions have been found. Channels were invented; a channel is a piece of hardware dedicated to the task of regulating the information traffic between the store and the peripheral to which it is attached, and doing this synchronized to the natural speed of the peripheral device, thus doing away with the implicit mutual synchronization of the peripheral devices that would be caused if both were controlled by the same sequential program execution.

The introduction of channels created two problems, a microscopic and a macroscopic one. The microscopic problem has to do with access to the store. In the old arrangement only the central processor required access to the store and when the central processor required access to the store it could get it. In the new arrangement, with the channels added—channels that can be regarded as "special purpose processors"—a number of processors can be competing with eachother as regards access to the store because such accesses from different processors very often exclude eachother in time (for technical or logical reasons). This microscopic problem has been solved by the invention of the "switch", granting the competing processors access to the store according to some priority rule. Usually the channels have a lower traffic density and a higher priority than the central processor: the processor works at full speed until a channel requests access to the store, an arrangement which is called "cycle stealing". We draw attention to the fact that the unit of information in which this interleaving takes place—usually "a word"—is somewhat arbitrary; in a few moments we shall encounter a similar arbitrariness.

The macroscopic problem has to do with the coordination of central processor activity and channel activity. The central processor issues a command to a channel and from that moment onwards, two activities are going on simultaneously and—macroscopically speaking—independent of eachother: the central processor goes on computing and the channel transport information. How does the central processor discover, when the execution of the channel command has been completed? The answer to this has been the "interrupt". Upon completion of a channel command the channel sets an interrupt flip-flop; at the earliest convenient moment, (but never sooner that after completion of the current instruction) the central processor interrupts the execution of the current program (in such a neat way that the interrupted computation can be resumed at a later moment as if nothing has happened) and starts executing an interrupt program instead, under control of which all now appropriate actions will be taken. From the point of view of the central processor it interleaves the various program executions, the unit of interleaving being—similarly arbitrarily—"the instruction".

The above scheme can be recognized in all larger, modern computers that I ever studied. It has been embellished in many directions but we don't need to consider those embellishments now. We go immediately to the next questions: given a piece of equipment constructed along the lines just sketched, what are the problems when we try to use it and in what direction should we look for their solution?

What are the problems? Well the main point is that from the point of view of program control such a piece of equipment must me regarded as a non-deterministic machine. Measured in a grain of time appropriate for the description of the activity of the centrol processing unit—clockpulse or instruction execution time—the time taken by a peripheral transport must be regarded as undefined. If completion of such a peripheral is signalled to the central processor by means of an interrupt, this means that we must regard the moment when the interrupt will take place (or more precisely: the point of progress where the computation will be interrupted) as unpredictable. The problem is that in spite of this indeterminacy of the basic hardware, we must make a more or less deterministic automaton out of this equipment: from the outside world the machine will be confronted with a well-defined computational task and it has to produce a well-defined result in a microscopically unpredictable way!

Let me give a simple example to explain what I mean by "a more or less deterministic automaton". Suppose that offering a program to the machine consists of loading a pack of cards into a card reader (and pushing some button on the reader in order to signal that it has been loaded). Suppose now that we have a machine with two readers and that we want to load it with two programs, A and B, and that we can do this by loading both card readers and pressing both buttons. We assume that the two card readers are not mutually synchronized, i.e. we regard both speeds as unpredictable. To what extent will the total configuration be a deterministic automaton? It will be fully deterministic in the sense that eventually it will produce both output A and output B. If these outputs are to be produced by the same printer, they will be produced in some order and the system may be such that the order in which the respective outputs appear on the printer *does* depend on the relative speeds of the two readers. As far as the operator is concerned, who has to take the output from the printer and to dispatch it to the customers, the installation is non-deterministic: what *he* has to do depends on the unpredictable speed ratio of the two readers, which may cause output A to precede or to follow output B. For both cases the operator has his instructions such that in both cases all output is dispatched to the proper customer. The "computation centre"—i.e. installation and operator together—are deterministic. We can regard the operator's activity as an outer layer, "wrapping up the installation", shielding from the outside world a level of interior indeterminacy.

Now, even if the operator is aware of not having a fully deterministic machine, we should recognize that he has only to deal with two cases—output A before output B or the other way round—while the number of possible sequences of occurrences at cycle time level is quite fantastic. In other words, by far the major part of the "shielding of indeterminacy" is done by the installation itself. We call the resulting installation "more or less deterministic" because as the case may be, a few degrees of limited freedom—here one boolean degree of freedom—may be left unpredictable.

We have called the operator's activity "an outer layer", shielding a level of indeterminacy, and of course we did so on purpose. At the other end we may distinguish an inner layer, viz. in the channel signalling (via an interrupt signal) that the next card has been read: it tells the central processor that the next card image is available in core, regardless which storage cycles have been stolen to get it there. The terms "inner layer" and "outer layer" have been chosen in order to suggest that in the total organization we shall be able to distinguish many layers in between. But an important remark is immediately appropriate: I assume that with the card read command an area in core has been designated to receive this card image: the remark that the interrupt signalled the completed transfer of the card image irrespective of which cycles had been stolen to transport its constituents is only true, provided that no other access to the designated core area took place in the period of time ranging from the moment the command was given up to the moment that the completion was signalled! Obvious but vital.

It draws our attention to an element of structure that must be displayed by the remaining programs if we wish to make the total organization insensitive to the exact identity of the cycles stolen by the channel. And from the above it is clear that this insensitivity must be one of our dearest goals. And on next levels (of software) we shall have to invent similar elements of structure, making the total organization insensitive (or "as insensitive as possible") to the exact moment when interrupts are honoured. Again it is clear that this must be one of our dearest goals. And on a next level we must make our organization insensitive (or "as insensitive as possible") to the exact number of cards put into the readers for program A and B, and so on ... This "layered insensitivity" is, in two words, our grand plan.

I have used the term "layer" on purpose, because it has seemed to provide an attractive terminology in terms of which to talk about operating systems and their total task. We can regard an operating system as the basic software that "rebuilds" a given piece of hardware into a (hopefully) more attractive machine. An operating system can then be regarded as a sequence of layers, built on top of eachother and each of them implementing a given "improvement". Before going on, let me digress for a moment and let me try to explain why I consider such an approach of ordered layers a fruitful one.

There is an alternative approach, which I would like to call the approach via unordered modules. There one makes a long list of all the functions of the operating system to be performed, for each function a module is programmed and finally all these modules are glued together in the fervent hope that they will cooperate correctly and will not interfere disastrously with eachother's activity. It is such an approach which has given rise to the assumed law of nature, that complexity grows as the square of the number of program components, i.e. of the number of "functions".

In the layered approach we start at the bottom side with a given hardware machine A_0, we add our bottom layer of software rebuilding A_0 into the slightly more attractive machine A_1, for which the next layer of software is programmed rebuilding it into the still more attractive machine A_2 etc. As the machines in the sequence $A_0, A_1, A_2, \ldots$ get more and more attractive, adding a further layer gets easier and easier. This is in sharp contrast to the approach via unordered modules, where adding new functions seems to get progressively worse!

1.

So much in favour of a layered approach in general. When one wishes to design an operating system, however, one is immediately faced with the burning question, which "improvement" is the most suitable candidate to be implemented in the bottom layer.

For the purpose of this discussion I will choose a very modest bottom layer. I do so for two reasons. Firstly, it is a choice with which for historical reasons I myself am most familiar. Secondly, as a bottom layer it is very modest and neutral, so neutral in fact that it provides us with a mental platform from where we can discuss various alternatives for the structure of what is going to be built on top of it. As a bottom layer it seems close to the choice of minimal commitment. The fact that this bottom layer is chosen as a starting point for our discussion is by no means to be interpreted as the suggestion that this is the best possible choice: on the contrary, one of the later purposes of this discussion is the consideration of alternatives.

With the hardware taking care of the cycle stealing we felt that the software's first responsibility was to take care of the interrupts, or, to put it a little more strongly, to do away with the interrupt, to abstract from its existence. (Besides all rational arguments this decision was also inspired by fear based on the earlier experience that, due to the irreproducibility of the interrupt moments, a program bug could present itself misleadingly like an incidental machine malfunctioning.) What does it mean "to do away with the interrupt"? Well, without interrupt the central processor continues the execution of the current sequential process while it is the function of the interrupt to make the central processor available for the continuation of another sequential process. We would not need interrupt signals if each sequential process had its own dedicated processor. And here the function of the bottom layer emerged: to create a virtual machine, able to execute a number of sequential programs in parallel as if each sequential program had its own private processor. The bottom layer has to abstract of the existence of the interrupt or, what amounts to the same thing, it has to abstract from the identity of the single hardware processor. If this abstraction is carried out rigorously it implies that everything built on top of this bottom layer will be equally applicable to a multiprocessor installation, provided that all processors are logically equivalent (i.e. have the same access to main memory etc.). The remaining part of the operating system and user programs together then emerges as a set of harmoniously cooperating sequential processes.

The fact that these sequential processes out of the family have to cooperate harmoniously implies that they must have the means of doing so, in particular, they must be able to communicate with eachother and they must be able to synchronize their activities with respect to eachother. For reasons which, in retrospect, are not very convincing, we have separated these two obligations. The argument was that we wished to keep the bottom layer as modest as possible, giving it only the duty of processor allocation; in particular it would leave the "neutral, monotonous memory" as it stood, it would not rebuild that part of the machine and immediately above the bottom layer the processes could communicate with eachother via the still available, commonly accessible memory.

The mutual synchronization, however, is a point of concern. Closely, related to this is the question: given the bottom layer, what will be known about the speed ratios with which the different sequential processes progress? Again we have made the most modest assumption we could think of, viz. that they would proceed with speed ratios, unknown but for the fact that the speed ratios would differ from zero. I.e. each process (when logically allowed to proceed, see below) is guaranteed to proceed with some unknown, but finite speed. In actual fact we can say more about the way in which the bottom layer grants processor time to the various candidates: it does it "fairly" in the sense that in the long run a number of identical processes will proceed at the same macroscopic speed. But we don't tell, how "long" this run is and the said fairness has hardly a logical function.

This assumption about the relative speeds is a very "thin" one, but as such it has great advantages. From the point of view of the bottom layer, we remark that it is easy to implement: to prevent a running program to monopolize the processor an interrupting clock is all that is necessary. From the point of view of the structure built on top of it is also extremely attractive: the absence of any knowledge about speed ratios forces the designer to code all synchronization measures explicitly. When he has done so he has made a system that is very robust in more than one sense.

Firstly he has made a system that will continue to operate correctly when an actual change in speed ratios is caused, and this may happen in a variety of ways. The actual strategy for processor allocation as implemented by the bottom layer, may be changed. In a multiprocessor installation the number of active processors may change. A peripheral may temporarily work with speed zero, e.g. when it requires operator attention. In our case the original line printer was actually replaced by a faster model. But under all those changes the system will continue to operate correctly (although perhaps not optimally, but that is quite another matter).

Secondly—and we shall return to this in greater detail—the system is robust thanks to the relative simplicity of the arguments that can convince us of its proper operation. Nothing being guaranteed about speed rations means that in our understanding of the structure built on top of the bottom layer we have to rely on discrete reasoning and there will be no place for analog arguments, for other purposes than overall justification of chosen strategies. I trust the strength of this remark will become apparent as we proceed.

2.

Let us now focus our attention upon the synchronization. Here a key problem is the so-called "mutual exclusion problem". Given a number of cyclic processes of the form

```
cycle begin entry;
            critical section;
            exit;
            remainder of cycle
      end
```

program "entry" and "exit" in such a way that at any moment at most one of the processes is engaged in its critical section. The solution must satisfy the following requirements:

a) The solution must be symmetrical between the processes; as a result we are not allowed to introduce a static priority.

b) Nothing may be assumed about the ratio of the finite speeds of the processes; we may not even assume their speeds to be constant in time.

c) If any of the processes is stopped somewhere in "remainder of cycle", this is not allowed to lead to potential blocking of any of the others.

d) If more than one process is about to enter its critical section, it must be impossible to devise for them such finite speeds, that the decision to determine which of them will enter its critical section first is postponed until eternity. In other words, constructions in which "After you"—"After you"—blocking, although improbable, is still possible, are not to be regarded as valid solutions.

I called the mutual axclusion problem "a key problem". We have met something similar in the situation of programs A and B producing their output in one of the two possible orders via the same printer: obviously those two printing processes have to exclude eachother mutually in time. But this is a mutual exclusion on a rather macroscopic scale and in all probability it is not acceptable that the decision to grant the printer to either one of the two activities will be taken on decount of the requirement of mutual exclusion alone: in all probability considerations of efficiency or of smoothness of service require a more sophisticated printer granting strategy. The explanation why mutual exclusion must be regarded as a key problem must be found at the microscopic end of the scale. The switch granting access to store on word basis provides a built in mutual exclusion, but only on a small, fixed and rather arbitrary scale. The same applies to the single processor installation which can honour interrupts in between single instructions: this is a rather arbitrary grain of activity. The problem arises when more complicated operations on common data have to take place. Suppose that we want to count the number of times something has happened in a family of parallel processes. Each time such an occurrence has taken place, the program could try to count it via

$$“n := n + 1”.$$

If in actual fact such a statement is coded by three instructions

```
"$R := n$;
 $R := R + 1$;
 $n := R$"
```

then one of the increases may get lost when two such sequences are executed, interleaved on single instruction basis. The desire to compound such (and more complicated) operators on commom variables is equivalent to the desire to have more explicit control over the degree of interleaving than provided by the neutral, standard hardware. This more explicit control is provided by a solution to the mutual exclusion problem.

We still have to solve it. Our solution depends critically on the communication facilities available between the individual processes and the common store. We can assume that the only mutual exclusion provided by the hardware is to exclude a write instruction or a read instruction, writing or reading a single word. Under that assumption the problem has been solved for two processes by T. J. Dekker in the early sixties. It has been solved by me for N processes in 1965 (C.A.C.M., 1965, Vol. 8, nr. 9, p. 569). The solution for two processes was complicated, the solution for N processes was terribly complicated. (The program pieces for "enter" and "exit" are quite small, but they are by far the most difficult pieces of program I ever made. The solution is only of historical interest.)

It has been suggested that the problem could be solved when the individual processes had at their disposal an indivisible "add to store" which would leave the value thus created in one of the private process registers as well, so that this value is available for inspection if so desired. Indicating this indivisible operation with braces the suggested form of the parallel programs was:

```
cycle begin while $\{x := x + 1\} \neq 1$ do $\{x := x - 1\}$;
            critical section;
            $\{x := x - 1\}$;
            remainder of cycle
      end.
```

Where the "add to store" operation is performed on the common variable "x" which is initialized with the value zero before the parallel grograms are started.

As far as a single process is concerned the cumulative Δx as effected by this process since its start is $=0$ or $=1$; in particular, when a process is in its critical section, its cumulative $\Delta x = 1$. As a result we conclude that at any moment when N processes are in their critical section simultaneously, $x \geqq N$ will hold.

A necessary and sufficient condition for entering a critical section is that *this* process effectuates for x the transition from 0 to 1. As long as one process is engaged in its critical section ($N = 1$), $x \geqq 1$ will hold. This excludes the possibility of the transition from 0 to 1 taking place and therefore no other process can enter its critical section. We conclude that mutual exclusion is indeed guaranteed. Yet the solution must be rejected: it is not difficult to see that even with two processes (after at least one succesful execution of a critical section) "After you"—"After you"—blocking may occur (with the value of x oscillating between 1 and 2).

A correct solution exists when we assume the existence of an indivisible operation "swap" which causes a common variable (x) and a private variable (loc) to exchange their values. With initially $x = 0$ the structure of the parallel programs is:

```
begin integer loc; loc := 1;
      cycle begin repeat swap (x, loc) until loc = 0;
                  critical section;
                  swap (x, loc);
                  remainder of cycle
            end
end.
```

The invariant relation is that of the $N+1$ variables (i.e. the N loc's and the single x) always exactly one will be $=0$, the others being $=1$. A process is in its critical section if and only if its own loc $=0$, was a result at most one process can be engaged in its critical section. When none of the processes is in its critical section, $x = 0$ and "After you"—"After you"—blocking is impossible. So this is a correct solution.

In a multiprogramming environment, however, the correct solutions referred to or shown have a great drawback: the program section called "enter" contains a loop in which the process will cycle when it cannot enter its critical section. This so-called "busy form of waiting" is expensive in terms of processing power, because in a multiprogramming environment (with more parallel processes than processing units) there is a fair chance that there will be a more productive way of spending processing power than giving it to a process that, to all intents any purposes, could go to sleep for the time being.

If we want to do away with the busy form of waiting we need some sort of synchronizing primitives by means of which we can indicate those program points where—depending on the circumstances—a process may be put to sleep. Similarly we must be able to indicate that potential sleepers may have to be woken up. What form of primitives?

Suppose that process 1 is in its critical section and that process 2 will be the next one to enter it. Now there are two possible cases.

a) process 1 will have done "exit" before process 2 has tried to "enter"; in that case no sleeping occurs

b) process 2 tries to "enter" before process 1 has done "exit"; in that case process 2 has to go sleep temporarily until is woken up as a side-effect of the "exit" done by process 1.

When both occurrences have taken place, i.e. when process 2 has succesfully entered its critical section it is no longer material whether we had case a) or case b). In that sense we are looking for primitives (for "enter" and "exit") that are commutative. What are the simplest commutative operations on common variables that we can think of? The simplest operation is inversion of a common boolean, but that is too simple for our purpose: then we have only one operation at our disposal and lack the possibility of distinguishing between "enter" and "exit". The next simplest commutative operations are addition to (and subtraction from) a common integer. Furthermore we observe that "enter" and "exit" have to compensate eachother: if only the first process passes its critical section the common state before its "enter" equals the common state after its "exit" as far as the mutual exclusion is concerned. The simplest set of operations we can think of are increasing and decreasing a common variable by 1 and we introduce the special synchronizing primitives

$$P(s): s := s - 1$$

and

$$V(s): s := s + 1$$

special in the sense that they are "indivisible" operations: if a number of P- and V-operations on the same common variable are performed "simultaneously" the net effect of them is as if the increases and decreases are done "in some order".

Now we are very close to a solution: we have still to decide how we wish to characterize that a process may go to sleep. We can do this by making the P- and V-operations operate not on just a common variable, but on a special purpose integer variable, a so-called *semaphore*, whose value is by definition non-negative; i.e. $s \geqq 0$.

With that restriction, the V-operation can always be performed: unsynchronized execution of the P-operation, however, could violate it.

We therefore postulate that whenever a process initiates a P-operation on a semaphore whose current value equals zero, the process in question will go to sleep until (another) process has performed a V-operation on that very same semaphore. A little bit more precise: if a semaphore value equals zero, one or more processes may be blocked by it, eager to perform a P-operation on it. If a V-operation is performed on a semaphore blocking a number of processes, one of them is woken up, i.e. will perform its now admissible P-operation and proceed. The choice of this latter process is such that no process will be blocked indefinitely long. A way to implement this is to decide that no two processes will initiate the blocking P-operation simultaneously and that they will be treated on the basis "first come, first served" (but it need not be done that way, see below).

With the aid of these two primitives the mutual exclusion problem is solved very easily. We introduce a semaphore "mutex" say, with the initial value

$$\text{mutex} = 1,$$

after which the parallel processes controlled by the program

```
cycle begin P (mutex);
            critical section;
            V (mutex);
            remainder of cycle
      end
```

are started.

Before proceeding with the discussion I would like to insert a remark. In languages specifically designed for process control I have met two other primitives, called "wait" and "cause", operating on an "event variable", which is a (possibly empty) queue of waiting processes. Whenever a process executes a "wait" it attaches itself to the queue until the next "cause" for the same event, which empties the queue and signals to all processes in the queue that they should proceed. Experience has shown that such primitives are very hard to use. The reason for this is quite simple: a "wait" in one process and a "cause" in another are non-commutative operations, their net effect depends on the order in which they take place and at the level where we need the synchronizing primitives we must assume that we have not yet effective control over this ordering. The limited usefulness of such "wait" and "cause" primitives could have been deduced a priori.

3.

As a next interlude I am going to prove the correctness of our solution. One may ask "Why bother about such a proof, for the solution is obviously correct". Well, in due time we shall have to prove the correctness of the implementation of more sophisticated rules of synchronization and the proof structure of this simple case may then act as a source of inspiration.

With each process "j" we introduce a state variable "C_j", characterizing the progress of the process.

$C_j = 0$ process$_j$ is in the "remainder of cycle"

$C_j = 1$ process$_j$ is in its "critical section".

While process$_j$ performs (i.e. "completes") the operation $P(\text{mutex})_j$ the transition $C_j = 0 \rightarrow C_j = 1$ takes place, when it performs the operation $V(\text{mutex})_j$ the transition $C_j = 1 \rightarrow C_j = 0$ takes place. (Note that the C_j are *not* variables occurring in the program, they are more like functions defined on the current value of the order counters.) In terms of the C_j the number of processes engaged in its critical section equals

$$\sum_{j=1}^{N} C_j .$$

In order to prove that this number will be at most $= 1$, we follow the life history of the quantity

$$K = \text{mutex} + \sum_{j=1}^{N} C_j .$$

The quantity K will remain constant as long as its constituents are constant: the only operations changing its constituents are the $2N$ mutually exclusive primitive actions $P(\text{mutex})_i$ and $V(\text{mutex})_i$ (for $1 \leqq i \leqq N$).

We have as a result of

$$\begin{aligned} P(\text{mutex})_i\colon \ \Delta K &= \Delta\,\text{mutex} + \Delta\left(\sum_{j=1}^{N} C_j\right) \\ &= \Delta\,\text{mutex} + \Delta\, C_i \\ &= -1 + 1 = 0 \end{aligned}$$

and similarly, as a result of

$$\begin{aligned} V(\text{mutex})_i\colon \ \Delta K &= \Delta\,\text{mutex} + \Delta\, C_i \\ &= +1 - 1 = 0 . \end{aligned}$$

As these $2N$ operations are the only ones affecting K's constituents, we conclude that K is constant, in particular, that it is constantly equal to its initial value,

$$K = 1 + \sum_{j=1}^{N} 0 = 1.$$

As a result

$$\sum_{j=1}^{N} C_j = 1 - \text{mutex}.$$

Because mutex is a semaphore, we have

$$0 \leqq \text{mutex},$$

and from the last two relations we conclude

$$\sum_{j=1}^{N} C_j \leqq 1.$$

Because this sum is the sum of non-negative terms we known

$$0 \leqq \sum_{j=1}^{N} C_j.$$

Combining this with

$$\text{mutex} = 1 - \sum_{j=1}^{N} C_j.$$

We conclude

$$\text{mutex} \leqq 1$$

i.e. mutex is a so-called "binary semaphore", only taking on the values 0 and 1.

Finally we observe that no process will be kept out of its critical section without justification: if all processes are outside their critical sections, all C_j's are $= 0$ and therefore mutex is $= 1$, thereby allowing the first process that wants to enter its critical section to do so.

For later reference we summarize the structure of this proof. A central role is played by an invariant relation among common variables (here only the semaphore) and "progress variables" (here the C_j's). Its invariance is proved by observing the net effect of the (mutually exclusive) operators operating on its constituents, without any further assumptions about their mutual synchronization, about which we can then make assertions on account of the established invariance. In the sequel we shall see that this pattern of proof is very generally applicable.

4.

Before proceeding with more complicated examples of synchronization we must make a little detour and make a connection with earlier observations. When a process is engaged in its critical section, a great number of other processes may go to sleep. When the first one leaves its critical section, it is undefined which of the sleepers is woken up, the only requirement being that no single process is kept sleeping indefinitely long. (This latter assumption we have to make when, later, we wish to prove assertions about the finite progress of individual processes.)

In this sense our "family of sequential processes" is still a mechanism of an undeterministic nature, but the degree of undeterminacy is a mild one compared with the original hardware, in which an interrupt could occur between any pair of instructions: the only indeterminacy left is the relative order of much larger units of action, viz. the critical sections. In this respect the bottom layer of our operating system achieves a step towards our goal of "layered insensitivity".

It is in this connection that I should like to make another remark of quantitative nature. The choice of the process to be woken up is left undefined because it is assumed that it does not matter, i.e. we assume the system load to be such that the total period of time that any of the processes will be engaged in its critical section will be a negligible fraction of real time, in other words, nearly always mutex = 1 will hold. It is for that reason that such a neutral policy for waking up a sleeper is permissible. This is no longer true for our macroscopic concerns regarding so-called "resource allocation". In the case of a number of programs producing their output via the same printer, these printing actions have to exclude eachother mutually in time, but it is no longer true that the total time spent in printing will be a negligible fraction of real time! On the contrary: in a well-balanced system the printer will be used with a duty cycle close to 100 percent! In order to achieve this—and to satisfy other, perhaps conflicting design requirements—such a neutral policy which is adequate for granting entrance into critical sections will certainly be inadequate for granting a scarce resource like a printer. For the implementation of a less neutral granting policy we shall use the critical sections, entrance to which is granted on a neutral basis. (For an example of a more elaborate synchronization implemented with the aid of critical sections we refer to the Problem of the Dining Philosophers to be treated later.) This is the counterpart of the "layered insensitivity": going upwards in levels we gain more and more control over the microscopic indeterminacy, but simultaneously macroscopic strategic concerns begin to enter the picture: it seems vital that the bottom layer with its microscopic concerns does not bother itself with such macroscopic considerations. This observation seems to apply to all well-designed systems: I would call it a principle if I had a better formulation for it.

5.

We now turn to a slightly more complicated example, viz. a bunch of producers and a bunch of consumers, coupled to eachother via an unbounded buffer. In this example all producers are regarded as equivalent to eachother and all consumers are regarded as quivalent to eachother. Under these assumptions—which are not very realistic—the semaphores provide us with a ready-made solution.

In the commonly accessible universe we have

a) a buffer, initialized empty

b) a semaphore "mutex", initialized = 1; this semaphore caters for the mutual exclusion of operations changing buffer contents

c) a semaphore "numqueuepor"; this gives (a lower bound of) the number of portions queueing in the buffer.

Then a producer may have the form

```
cycle begin produce next portion;
            P(mutex);
            add portion produced to buffer;
            V(numqueuepor);
            V(mutex)
      end
```

with consumers of the following structure

```
cycle begin P(numqueuepor);
            P(mutex);
            take portion from buffer;
            V(mutex);
            consume portion taken
      end.
```

Note 1: The order of the *V*-operations in the producer is immaterial, the order of the *P*-operations in the consumer is absolutely essential.

Note 2: The assumption is that the operations "produce next portion" and "consume portion taken" are the slow, timeconsuming operations—possible in synchronism with other equipment—for which parallelism is of interest, while the actions "add portion produced to buffer" and "take portion from buffer" are very fast "clerical" operations.

In the above program the semaphore "numqueuepor" is a so-called "general semaphore", i.e. a semaphore whose possible values are not restricted to 0 and 1. We shall now give an alternative program, using only binary semaphores.

In the commonly accessible universe we have

a) a buffer and an integer "n", counting the number of portions in the buffer. The buffer is initialized empty (incl. $n := 0$)

b) a semaphore "mutex" initialized $= 1$; this semaphore caters for the mutual exclusion of the operations changing the buffer contents, the value of "n" and the inspection of "n".

c) a semaphore "consal", initialized $= 0$; if this semaphore is $= 1$, a next consumption is allowed.

Then a producer may have the form

```
cycle begin produce next portion;
            P(mutex);
            add portion to buffer (incl. $n := n + 1$);
            if $n = 1$ do V(consal);
            V(mutex)
      end
```

with consumers of the following structure

```
cycle begin P(consal);
            P(mutex);
            take portion from the buffer (incl. $n := n - 1$);
            if $n > 0$ do V(consal);
            V(mutex);
            consume portion taken
      end.
```

Although it is not too hard to convince ourselves "by inspection"—whatever that may mean— that the above bunch of programs work properly, it is illuminating to give a somewhat more formal treatment of their cooperation. (I am now used to calling such a more formal treatment of their cooperation "a correctness proof", although I did not formalize the requirements that such a piece of reasoning should satisfy in order to be a "valid proof".)

The proof consists of two steps. The first step uses our earlier result, viz. that the P(mutex) and V(mutex) establish mutual exclusion of the critical sections. (Inside these critical sections we find no P-operations, as a result they cannot give rise to deadlock situations.) This observation allows us to regard the critical sections as indivisible operations and to confine our attention to the state of the system at the discrete moments with mutex $= 1$ (i.e. no one engaged in its critical section).

In the second step we define three mutually exclusive states for the whole system and shall show that whenever the system is started in one of these states, it will remain within these states. For the purpose of state description we introduce a function defined on the progress of the consumers, viz.

$K =$ the number of consumers that have performed "P(consal)" but have not yet entered the following critical section.

Now we can introduce our three states

$$S1: n = 0 \quad and \quad K = 0 \quad and \quad \text{consal} = 0$$
$$S2: n > 0 \quad and \quad K = 0 \quad and \quad \text{consal} = 1$$
$$S3: n > 0 \quad and \quad K = 1 \quad and \quad \text{consal} = 0.$$

Three operations; (viz. P(consal) and the two critical sections) operate on the constituents of these boolean expressions; for each state we investigate all three.

$S1$: (initial state)
P(consal) : impossible (on account of consal $= 0$)
critical producer section : transition to $S2$
critical consumer section : impossible (on account of $K = 0$)

$S2$:
P(consal) : transition to $S3$
critical producer section : transition to $S2$
critical consumer section : impossible (on account of $K = 0$)

$S3$:
P(consal) : impossible (on account of consal $=0$)
critical producer section : transition to $S3$
critical consumer section : transition to $S1$ or $S2$.

This concludes the second step, showing the invariance of

$$S1 \text{ or } S2 \text{ or } S3$$

(from which we conclude $N \geqq 0$ and consal $\leqq 1$).

A few remarks, however, are in order, for we have cheated slightly. Let us repair our cheating first and then give our further comments. In our second step we have investigated the *isolated* effect of either P(consal) or the critical producer section or the critical consumer section. For the critical sections this is all right for they exclude eachother mutually in time; the operation P(consal), however, can take place *during* a critical section, and we did not pay any attention to such coincidence. We can save the situation by observing that in the case of coincidence the net effect is equal to the execution of the critical section immediately followed by P(consal). This is really a messy patching up of a piece of reasoning that was intended to be clean. Now our further comments.

1) The proof shows why the mutual exclusion problem is worthy of the name "a key problem". Thanks to the mutual exclusion of critical sections we only need to consider the net effect of each single, isolated section. If these sections were not critical, i.e. could take place in arbitrary interleaving, we would have to consider the net effect of one section, the net effect of two sections together, of three sections together, of four etc.! With N cooperating processes the number of cases to be investigated would grow like 2^N (i.e. the powerset!). This is one of the strongest examples showing how the amount of intellectual effort needed for a correctness proof may depend critically on structural aspects of the program, here the aspect of mutual exclusion. It is this observation that is meant to justify the inclusion of the above proof in this text.

2) The proof is complicated considerably by the fact that P(consal) is an operation sequentially separate from the following critical section: this caused the messy patching up of our piece of reasoning, it called for the introduction of the function "K". If the conditional entrance of critical sections is going to be a standard feature of the system, a more direct way of expressing this would be essential. A minimal departure of the current formation would be the introduction of the parallel P-operation, allowing us to combine the two P-operations of the consumer into

$$P(\text{consal, mutex}).$$

3) For the sake of completeness we mention that in the THE multiprogramming system, were we used general semaphores to control synchronization along information streams, each information stream had at any moment in time at most one consumer attached to it. As a result a general semaphore could block at most one process and when a V-operation was performed on it there was never the problem which process should be woken up. The absence of the possibility that more than one process is blocked by a general semaphore is not surprising:

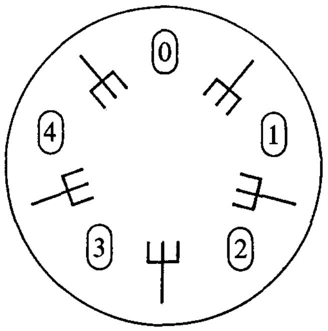
it is the semaphore "consal" that may be equal to zero for a long period of time; as a result it is not to be expected that it is irrelevant which of the processes will be woken up when a *V*-operation is performed on it. In the design phase of the THE multiprogramming system the parallel *P*-operation has been considered but finally it has not been implemented because we felt that it contained the built-in solution to an irrealistic problem. But it would have simplified proof procedures.

6.

We now turn to the problem of the Five Dining Philosophers. The life of a philosopher consists of an alternation of thinking and eating:

```
cycle begin think;
            eat
      end.
```

Five philosophers, numbered from 0 through 4 are living in a house where the table is laid for them, each philosopher having his own place at the table:



Their only problem—besides those of philosophy—is that the dish served is a very difficult kind of spaghetti, that has to be eaten with two forks. There are two forks next to each plate, so that presents no difficulty: as a consequence, however, no two neighbours may be eating simultaneously.

A very naive solution associates with each fork a binary semaphore with the initial value $= 1$ (indicating that the fork is free) and, naming in each philosopher these semaphores in a local terminology, we could think the following solution for the philosopher's life adequate

```
cycle begin think;
            P(left hand fork); P(right hand fork);
            eat;
            V(left hand fork); V(right hand fork)
      end.
```

But this solution—although it guarantees that no two neighbours are eating simultaneously—must be rejected because it contains the danger of the deadly embrace. When all five philosophers get hungry simultaneously, each will grab his left hand fork and from that moment onwards the group is stuck. This could be overcome by the introduction of the parallel P-operation, combining the two P-operations into the single

$$P(\text{left hand fork, right hand fork}).$$

For the time being we assume the parallel P-operation denied to us—later we shall reject the solution using it on other grounds—and we shall show how (using only single P-operations and binary semaphores) we can derive our solution in a reasonably controlled manner.

In order to be able to give a formal description of our restriction, we associate with each philosopher a state variable, "C" say, where

$C[i] = 0$ means: philosopher i is thinking

$C[i] = 2$ means: philosopher i is eating.

In accordance with their first act, all C's will be initialized $= 0$. In terms of the C's we can state that it is idsallowed

$$\exists_i \; (C[i] = 2 \; \textit{and} \; C[(i+1) \; \textit{mod} \; 5] = 2), \tag{1}$$

in words: no philosopher may be eating while his left hand neighbour is eating as well. From this formula it *follows* that for a C the transition from 2 to 0 can never cause violation of the restriction (1), while the transition from 0 to 2 can. *Therefore* we introduce for the last transition an intermediate state

$C[i] = 1$ means: philosopher i is hungry.

Now each philosopher will go cyclically through the states 0, 1, 2, 0 ... The next question to ask is: when has the (dangerous) transition from 1 to 2 to take place for philosopher K? Well, three conditions have to be satisfied

1) $C[K] = 1$, i.e. he himself must be hungry

2) $C[K+1) \; \textit{mod} \; 5] \neq 2$, because otherwise
$C[K] := 2$ would cause violation of (1) for $i = K$

3) $C[K-1) \; \textit{mod} \; 5] \neq 2$, because otherwise
$C[K] := 2$ would cause violation of (1) for $i = (K-1) \; \textit{mod} \; 5$.

As a result we have to see to it that the state

$$\exists_K (C[(K-1) \; \textit{mod} \; 5] \neq 2 \; \textit{and} \; C[K] = 1 \; \textit{and} \; C[(K+1) \; \textit{mod} \; 5] \neq 2) \tag{2}$$

is unstable: whenever it occurs, it has to be resolved by assigning $C[K] := 2$ and sending philosopher K to the table.

In a similar analysis we ask: which transitions in the life of philosopher w can cause the unstable situation and for which values of K?

1) when $C[w] := 1$ is executed, instability may be created for $K = w$

2) when $C[w] := 0$—i.e. when $C[w]$ loses the value 2—instability may be created for $K = (w+1) \; \textit{mod} \; 5$ and for $K = (w-1) \; \textit{mod} \; 5$.

In words: when philosopher w gets hungry, the test whether he himself should be sent to table is appropriate, when he leaves the table the test should be done for both his neighbours.

In the universe we assume declared

1) the *semaphore* mutex, initially $= 1$

2) the *integer array* $C[0:4]$, with initially all element $= 0$

3) the *semaphore array* prisem $[0:4]$ with initially all elements $= 0$

4) *procedure* test (*integer value* K);

```
    if C[(K - 1) mod 5] ≠ 2 and C[K] = 1 and C[(K + 1) mod 5] ≠ 2 do
                    begin C[K] := 2; V(prisem [K]) end;
```

(This procedure, which resolves unstability for K when present, will only be called from within a critical section).

In this universe the life of philosopher w can now be coded

```
cycle begin think;
            P(mutex);
               C[w] := 1; test (w);
            V(mutex);
            P(prisem [w]); eat
            P(mutex);
               C[w] := 0; test [(w + 1) mod 5]; test [(w - 1) mod 5];
            V(mutex)
      end.
```

And this concludes the solution I was aiming at. I have shown it, together with the way in which it was derived, for the following reasons.

1) The arrangement with the private semaphore for each process and the common semaphore for mutual exclusion in order to allow for unambiguous inspection and modification of common state variables is typical for the way in which in the THE multiprogramming system all synchronization restrictions have been implemented that were more complicated than straightforward mutual exclusion or synchronization along an information stream (the latter synchronization has been implemented directly with the aid of a general semaphore).

2) The solution (inclusive the need for the introduction of the intermediate state called "hungry") has been derived by means of a formal analysis of the synchronization restriction. It is exemplar for the way in which the flows of mutual obligations for waking up have been derived in the design phase of the THE multiprogramming system. It is this analysis that I have called "A constructive approach to the problem of program correctness".

With respect to this particular solution I would like to make some further remarks.

Firstly the solution as presented is free from the danger of deadlock, as it should be. Yet it is highly improbable that a solution like this can be accepted because it contains possibility of a particular philosopher being starved to death by a conspiration of his two neighbours. This can be overcome by more sophisticated rules (introducing besides the state "hungry" also the state "very hungry"); this requires a more complicated analysis but by and large it follows the same pattern as the derivation shown. This was another reason not to introduce the parallel P-operation: for the solution with the parallel P-operation we did not see an automatic way of avoiding the danger of individual starvation.

Secondly we could have made a more crude solution: the procedure "test" has a parameter indicating for which philosopher the test has to be done; also in the critical sections we call the procedure "test" precisely for those philosophers for whom there is a chance that they should be woken up and for no others. This is very refined: we could have made a test procedure without parameter that would simply test for any K if there was an unstability to be removed. But the problem could have been posed for 9 or 25 philosophers and the larger the number of philosophers, the more prohibitive the overhead of the crude solution would get.

Thirdly, I have stated that we "derived our solution in a reasonably controlled manner": although the formal analysis has been carried out almost mechanically, I would not like to suggest that it should be done automatically, because in real life, whether we like it or not, the situation can be more complicated.

We consider two classes of processes, class A and class B, sharing the same resource from a large pool. (The situation occurred in the THE multiprogramming system with the total pool of pages in the system.) Suppose now that processes from class A ask and return items from this pool at high frequency, while those from class B do so at low frequency only. In that case it is highly unattractive to pose upon the highly frequent item releases of class A the (possibly) considerable overhead involved in the analysis of whether it is necessary to wake up one or more blocked processes. This high-frequency overhead was avoided by delegating the waking-up obligation to (some) processes of class B and by garanteeing that at least one of these processes would be active when the boundary of the resource restriction was in danger of being approached. In other wirds, in order to reduce system overhead we removed the highly frequent inspection whether processes had to be woken up at the price of increasing the "reaction time" there where an ultra short "response" was not required. The taking of such decisions seems a basic responsibility of the system designer and I don't see how they could be taken automatically.

The above concludes my discussion of the chosen bottom layer. In the final part of this paper I would like to discuss briefly an alternative solution.

7.

The chosen bottom layer implements a family of sequential processes plus a few synchronizing primitives, the remaining part of the system, to be composed on top of it, will exist of a set of harmoniously cooperating sequential processes. The interface is characterized by a number of features

a) the bottom layer treats all sequential processes on the same footing

b) the sequential processes communicate to eachother via commonly accessible variables

c) critical sections ensure the unambiguous interpretation and modification of these common variables.

One or two objections can be raised to this organisation; they center around the observation that each sequential process can be in one of two mutually exclusive, radically different states: either the process is inside its critical section or it is not. Inside its critical section it is allowed to access the common variables, outside it is not. In actual fact this difference does not only pertain to accessibility of information, it has also a bearing on processor allocation as implemented in the bottom layer. Given a process without hurry it is permissible to take the processor away from it for longer periods of time, but it is unattractive to do so in the middle of a critical section: if a process is stopped within a critical section it blocks for the other processes the mechanism needed for their cooperation and the remaining processes are bound to come to a grinding halt. In the THE multi-programming system this has been overcome by giving processes two colours—red or white—by making each process red while it is in a critical section and by never granting the processor to a white process if a red one is logically allowed to proceed.

Furthermore there is the aspect of reproducibility. To an individual user, offering a strictly sequential program to the system, we should like to present a strictly deterministic automation. In the system a number of sequential processes are dedicated to the processing of user programs, they act as slots into which a user program can be inserted; whenever the user program refers to a shared resource the translator effectively inserts—via a subroutine call—the critical section required for this cooperation. As a result, what happens in this slot is perfectly reproducible as long as the sequential process remains outside critical sections. But if we wish to charge our user and also insist that the charge be reproducible, we can only charge him for the activity of the slot outside critical sections! What happens inside the critical sections is situation dependent system overhead: it does not really "belong" to the activity of the process in which the critical section occurs.

Finally, we know how to interpret the evolution of a sequential process as a path through "its" state space as is spanned by "its" variables. But for this interpretation to be valid, it is necessary that all variables "belong" uniquely to one sequential process.

It is this collection of observations that was an incentive to redo some of our thinking about sequential processes and to reorder the total activity taking place in the system. Instead of N sequential processes cooperating in critical sections via common variables, we take out the critical sections and combine them into a $N+1$st process, called "a secretary"; the remaining N processes are called "directors". Instead of N equivalent processes, we now have N directors served by a common secretary. (We have used the metaphor of directors and a common secretary because in the director-secretary relation in real-life organisation its also unclear who is the master and who is the slave!)

What used to be critical sections in the N processes are in the directors "calls upon the secretary".

The relation between a set of directors and their common secretary shows great resemblance to the relation between a set of mutually independent programs and a common library. What is regarded as a single, unanalysed action on the level of a director, is a finite sequential process on the level of the secretary, similar to the relation between main program and subroutines.

But there is also a difference. In the case of a common library of re-entrant procedures, the library does not need to have a private state space: whenever a library procedure is called its local state space can be embedded (for the duration of the call) in the (extendable) state space of the calling program.

A secretary, however, has her own private state space, comprising all "common variables". One of the main reasons to introduce the concept of "a secretary" is that now we have identified a process to which the "common variables" belong: they belong to the common secretary.

To stress the specific nature of a secretary, I call her "a semi-sequential process". A fully sequential process consists of a number of actions to be performed one after the other in an order determined by the evolution of this process. A secretary is a bunch of actions—"operators in her state space"—to be performed one after the other, but in an undefined order, i.e. depending on the calls of her directors.

A secretary presents itself primarily as a bunch of non-reentrant routines with a common state space. But as far as the activity of the main program is concerned there is a difference between the routine of a secretary and a normal subroutine. During normal subroutine call we can regard the main program "asleep", while the return from the subroutine "wakes" the main program again. When a director calls a secretary—for instance when a philosopher wishes to notify the secretary that now he is hungry—the secretary may decide to keep him asleep, a decision that implies that she should wake him up in one of her later activities. As a result the identity of the calling program cannot remain anonymous as in the case of the normal subroutine. The secretary must have variables of the type "process identity" whenever she is called the identity of the calling process is handed over in an implicit input parameter, when she signals a release—analogous to the return of the normal subroutine—she will supply the identity of the process to be woken up.

In real time a director can be in three possible states with respect to his secretaries

a) "active", i.e. his progress is allowed

b) "calling", i.e. he has tried to initiate a call on a secretary, but the call could not be honoured, e.g. because the secretary was busy with another call

c) "sleeping", i.e. a call has been honoured but the secretary's activity in which he will be released has not ended.

The state "calling" has hardly any logical significance: it would not occur if the director was stopped just before the call that could not be honoured.

With respect to her directors a secretary can be

a) "busy", i.e. engaged in one for her (finite) algorithms

b) "idle", i.e. ready to honour a next call from one of her directors.

Note that a secretary may be simultaneously busy with respect to her directors and calling or sleeping with respect to one of her subsecretaries.

In two respects, the above scheme asks for embellishments. Firstly, a secretary may be in such a state that certain calls on her service are inconvenient. With each call we can associate a masking bit, stating whether with respect to that call she is "responding" or "deaf". A secretary managing an unbounded buffer could be deaf for the consumer's call when her buffer is empty. Here we have another reason why a director may be in the state "calling": besides being busy the secretary could be deaf for the call concerned. For the reasons stated I have my doubts as to whether this embellishment is very useful, but I mention it because it seems more useful than similar embellishments that have been suggested, e.g. making a secretary responding to an enumerated list of directors. The secretary has to see to it that certain constraints will not be violated, i.e. she may be in such a state that she can not allow certain of her possible *actions* to take place. This has nothing to do with the identity of the director calling for such an action.

A more vital embellishment is parameter passing: in general a director will like to send a message to his secretary when calling her—a producing director will wish to hand over the portion to be buffered; in general a director will require an answer back from his secretary when she has released his call—a consuming director will wish to receive the portion to be unbuffered.

Note that this message passing system is much more modest than various mail box systems that have been suggested in which processes can send messages (and proceed!) to other processes. In such systems elaborate message queues can be built up. Such systems suffer from two possible drawbacks. Firstly, implementation reasons are apt to impose upper limits to lengths of message queues: "message queue full" may be another reason to delay a process and to show the absence of the danger of deadly embraces may prove to be very difficult. Secondly, and that seems worse, with the queueing messages we have reintroduced state information that cannot be associated with an individual process.

From an esthetic point of view the relation director-secretary is very pleasing because it allows secretaries to act as directors with respect to subsecretaries. This places our processes in a hierarchy which avoids deadly embraces as far as mutual exclusion is concerned in exactly the same way in which mutual exclusion semaphores would need to be ordered in the case of nested critical sections. Whether, however, actual systems can be built up with a meaningful hierarchy of secretaries of reasonable depth—say larger than two—remains to be seen. That is why I called this point of view "esthetically pleasing".

Finally: I can only view a well-structured system as a hierarchy of layers and in the design process the interface between these layers has to be designed and decided upon each time. I am not so much bothered by designer's willingness and ability to propose such interfaces, I am seriously bothered by the lack of commonly accepted yardsticks along which to compare and evaluate such proposals. My "playing" with a bottom layer should therefore not be regarded as a definite proposal for yet another interface, it was meant to illustrate a way of thinking.

Acknowledgement is due to my former students J. Bomhoff and W. H. J. Feyen and to Professor C. A. R. Hoare from the Queen's University of Belfast.

References

Koestler, A.: The act of creation. New York: McMillan 1970.
Simon, H. A.: The sciences of artifical. Cambridge: MIT Press 1969.

Prof. Dr. E. W. Dijkstra
Technische Hogeschool
Postbus 513
Eindhoven
Nederland