Over de sequentialiteit van procesbeschrijvingen
About the sequentiality of process descriptions*

Edsger W. Dijkstra

1 Introduction

It is not unusual for a speaker to start a lecture with an introduction. As some in the audience might be totally not familiar with the issues that I wish to address and the terminology, which I will have to use, I will to give two examples as a means of introduction, the first one to describe the background of the problem and a second one, to give you an idea of the kind of logical problems, that we will encounter.

1.1 Over-determination

My first introduction is a historical one. Firstly I want you to understand what a sequential process description is. An example of a sequential process description that you all know is the construction of the altitude from the vertex C of the triangle ABC. This description might read as follows:

1. draw the circle with centre A and radius = AC;
2. draw the circle with centre B and radius = BC;
3. draw the line, which is determined by the intersections of the circles mentioned under (1) and (2).

This description is for the benefit of those unfamiliar with the construction of the altitude line; for their benefit the construction of the altitude line consists of a number of actions of a more limited repertoire; actions, which may be executed in the specified order, one after the other, i.e. sequentially. It will not have escaped the attentive reader, that here we have specified more than strictly necessary: it is essential for action (3) to be carried out only after operations (1) and (2) have been completed; operations (1) and (2) may however be carried out in reverse order, or even concurrently by a drawer who can operate two compasses, lets say one in each hand, executing actions (1) and (2) simultaneously. But this freedom has disappeared entirely from our process description.

Now for a more numeric example, something more in line with the area, where the problems soon be treated have become actual. Given the values of the variables a, b, c and d, we are asked to calculate the value of the variable x, given by the formula $x \leftarrow (a + b) * (c + d)$. When performing this value assignment is not part of the primitive repertoire of the calculator — and you may by now start thinking of a calculator machine —, then this calculation rule should be constructed from more elementary steps such as:

*Translated from [1] by Martien van der Burgt and Heather Lawrence [2]. Edited by Matthieu Perrin. No section title were present in the original paper. They were only added here for readability and must not be taken too seriously.

1. Calculate $t_1 \leftarrow a + b$;
2. Calculate $t_2 \leftarrow c + d$;
3. Calculate $x \leftarrow t_1 * t_2$.

This calculation rule contains exactly the same over-determination as our altitude construction: the order of the first two steps is irrelevant, they could even be carried out simultaneously. The above process description is representative of the execution mode of more traditional computing machines. The repertoire of elementary operations, because technical components are required for every operation, is finite and for financial reasons even very limited; these calculators owe their enormous flexibility to the fact, that arbitrarily complex algebraic expressions as illustrated above can be broken down in a sequence of these elementary operations. Inherent to this sequential description of the process is, that one specifies more than strictly necessary. This has not been a problem for some time, but in recent years this has changed. This is due to two reasons: changes in the structure of machines and the advent of a new class of problems, where these machines are to be used.

The classical sequential machine does "one thing at a time" and performs one function after another. For the sake of efficiency many specific devices were built for specific functions, but if you hold on to carrying out the different operations strictly sequential, suddenly the machine presents itself as N cooperating bodies, of which only one or two are working at any discrete moment. If N is large, this means that most of your machine most of the time stands idle and that is clearly not the intention.

The second reason is the advent of a new category of problems; computing machines were originally only used for fully predefined processes, and the machine could execute the process on its own at its own pace. As soon as one wants the information processing method to cooperate with some other process — e.g. a chemical process, which must be directed by the computing machine — then very different types of problems occur. Referring to our example of the calculation of $x \leftarrow (a + b) * (c + d)$, we no longer are in the situation that it does not matter, which sum is created first. If the data a, b, c and d are provided at unknown moments by the environment of the computing machine and it is the task of the machine, to deliver on the value of x as soon as possible, this means that the machine should initiate with a partial calculation as soon as the necessary information is available: which sum one wants to be calculated first, will depend on the sequence in which the data a, b, c and d become available. This was my first introduction and with it I hope to have given you an idea of how you should visualize a sequential process description, and why there is a need for less strict sequential process descriptions.

1.2 Liveness

Now my second introduction, which is meant to give you right from the start an idea of the logical problems, we will run into by leaving the strict sequentiality. Just as in an ordinary piece of algebra variables are identified with letters, generally "names", in the actions of information processing one has the continuous duty to identify, the duty to find the object defined by a name. Under circumstances one would like to have the selection process, which the teacher in the classroom has, when she says "Johnny, come to the black board." Assuming that the class as a whole pays attention and obeys, this process works flawlessly under the assumption that there is one and only one Johnny in the classroom. The particularity of this selection mechanism is that it works regardless of the number of children in the classroom.

Unfortunately this idyll cannot be implemented technically or at the most with flaws and we have a problem, which is more in line with reality, when children do not spontaneously respond to calling their name and Miss, when she wants Johnny to the board, is forced to take Johnny by the ear and drag him to

the board. If it is an old fashioned school, where every child has its own place in the classroom, then the teacher can, when at the beginning of the year she has put the children on a particular place, rename them for her own convenience, and replace the name "Johnny" — which for finding the boy is not very helpful — by an identification, which, according to its structure immediately defines which ear she should pull: she can call the boy "third row, second desk". This is essentially the technique that is used in classical automata by classical applications.

In modern applications one can not do this, and one must cope with such problems, as Miss has, when the children can choose where they sit. To make it not impossible for Miss, we assume that a name label has been pinned on each of the obstinate children. If the teacher now wants Johnny to the board, she has to look up Johnny. The teacher works sequentially, can only read a label at a time, and the teacher works systematically: she ordered the desks for her self uniquely in one way or another and if she wants Johnny, she visits the desks in this order, always asking herself "Are you Johnny?", if not, then she goes to the next desk.

This process works flawlessly, if we can guarantee that the children do not change positions during the walk of the teacher, if we can guarantee that the two processes "child localization" and "child permutation" are not performed simultaneously. Because also you understand that Johnny, as soon as he realises, that he is the wanted figure, sits down in the back of the classroom and when the teacher searched the front half of the class, he jumps quickly forward. We can refine the teacher and make her, if she has searched the whole class and did not find Johnny, start all over from the beginning, but this is no solution, because Johnny immediately would jump back again. Now you might remark that if those permutations were not executed with so much malice but only randomly, then in that case the expected time for locating Johnny remains finite, his freedom of jumping around did not even increase it. This remark, how hopeful, brings no solace. Those who have studied the "Theory of computability" know how much weight is given to a finite, i.e. guaranteed to end algorithm, and they will understand that a process that potentially spins indefinitely, is no attractive element. Worse is, that in reality it is often a question, whether we are allowed to consider the permutations of the children as random. If the probability is finite, that when the teacher has made her round through the class, the children are seated exactly as when the teacher began to search, this could really be the first period of a pure periodic phenomenon. Even if we accept the risk that we will never find Johnny, still her misery does not stop here, because in addition to the risk that she will not find Johnny, she runs the much greater risk that she drags the wrong child to the board, i.e. when the children between the moment that she satisfactorily ascertains "So, you are Johnny" and the moment that she, based on the findings, stretches out her hand to the corresponding ear, swiftly change places. In an information processing system this would be tantamount to a disaster. This is the end of my second introduction, which I hope gave you some idea of the kind of problems that we will encounter.

1.3 Organization

At the end of my first introduction I mentioned two reasons which were responsible for a growing interest in less strictly sequential process descriptions. I've only slightly touched on them and it is you may not have noticed that they were derived from almost opposite entourage. In the first case I mentioned the increasing complexity of machines, which now are composed of a number of more or less autonomous working parts with the assignment to work together to execute one process. In the second case, where we looked at the machine in a so-called "real-time application", then we had a view of a single device that had the possibility, in case of a change in the external urgency situation, to switch to the now most urgent of its possible tasks. These two packages, a cooperation of a number of machines in one process and conversely, one machine dividing its attention on a number of processes, have long been regarded as alien to one another. They have even invented different names for it, one called "parallel programming" and the other called "multi programming". You

should not ask me what's what, because I can not remember that, since having discovered that the logical problems, that they evoke by the non-sequentiality of the process definition, are in both cases exactly the same.

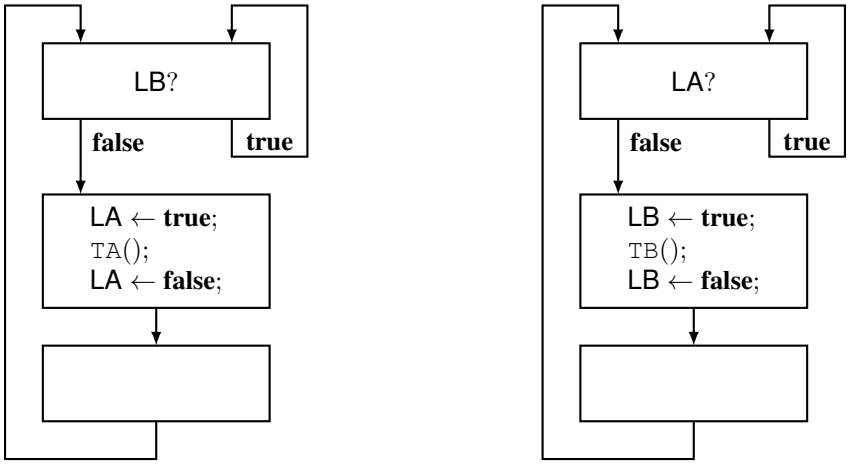
In the next part I will use the view of a number of mutually weakly coupled, in itself sequential machines, following the first entourage. But this is only a way of describing, because that what I will call a machine from now on is not necessarily a discrete identifiable piece of equipment; from now on my distinguished machines may be very abstract and they represent no more than a standalone sequential subprocess. This is partly reflected in my showing a predilection for a cooperation of N machines, N random. If I only used the term machine for a sensible functional piece of equipment, then I might consider the number of machines for each installation as fairly constant. Now that a machine, as personification of a standalone sequential subtask, may be created ad libitum by the user, such a fixed upper limit is no longer acceptable. So I'm going to talk about the cooperation of N machines, N not only arbitrary, but if it has to be, variable during the process. Superfluous machines can be destroyed, new machines can be created as needed and included in the cooperation of the others. I hope, that you will accept, now on my authority or afterwards by your own recognition, that we can limit ourselves without harming the generality to sequential machines, which execute some cyclical process.

2 The Mutual Exclusion Problem

2.1 Optimistic attempt

Let us start with a very simple problem. Given two machines A and B, both engaged in a cyclical process. In the cycle of machine A there is a certain critical section, called `TA`, and in that of machine B a critical section `TB`. The task is to make sure that never simultaneously both machines are each in their critical section. (In terms of our second introduction: A machine could be the teacher, trajectory `TA` the selection of a new student, to be called to the board, while the trajectory `TB` represents the process "child permutation".) There should be no assumptions made on the relative speeds of the machines A and B; the rate at which they work does not even need to be constant. It is clear that we can realize the mutual exclusion of the critical section only, if the two machines are in some way or another able to communicate with each other. For this communication, we establish some shared memory, i.e. a number of variables, which are accessible to both machines in either direction, i.e. to which both machines can assign a value and whose current value may be found out by both machines. These two actions, assigning a new value and inquiring about the current value are considered indivisible actions, i.e. if both machines wish to "simultaneously" assign a value to a common variable then the value assigned at the end is one or the other value, but not some mixture. Similarly, if one machine asks for the value of a shared variable at the time that the other machine is assigning a new value to it, then the requesting machine will receive or the old or the new value, not a random value. It will show that for this purpose we can limit ourselves to common logical variables, i.e. variables with only two possible values, which we following standard techniques denote with **true** or **false**. Furthermore, we assume that both machines can only refer to one common variable at one time and next only to assign a new value or to inquire after the current value. In Figure 1, a tentative solution is given in block diagram form for the structure of the program for both machines. Each block has its entrance at the top and its exit at the bottom. For a block with a dual output, the choice is determined by the value of the previously requested logical variable: has it the value **true**, then the output right intended, has a value of **false**, then we choose the exit left.

There are two common logical variables in this diagram, LA and LB. LA means machine A is in its critical section, LB means machine B is in its critical section. The schemata in Figure 1 are obvious. In the block top machine A waits if he arrives at a moment, that machine B is busy in section `TB`, until machine B has left



(a) Machine A

(b) Machine B

Figure 1: First attempt to solve mutual exclusion

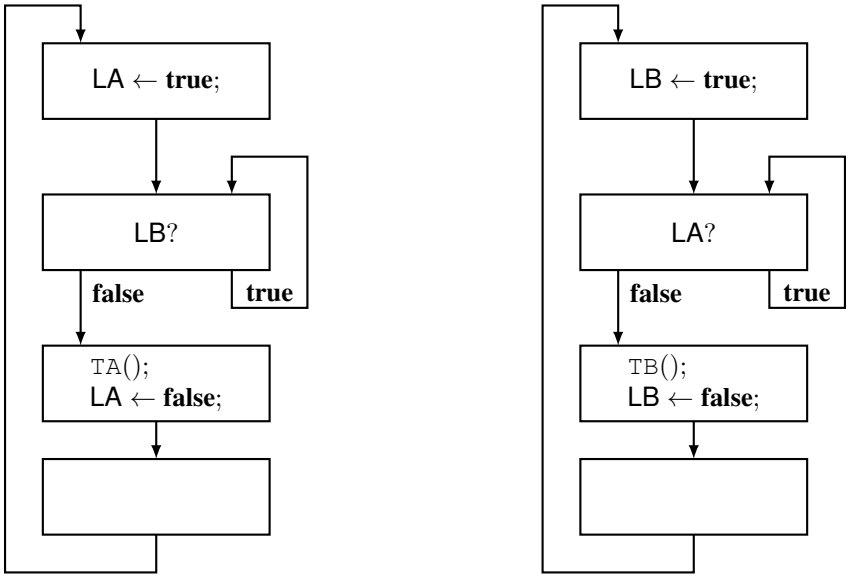
its critical section, which is marked by the assignment $LB \leftarrow$ **false**, which next lifts the wait of machine A. And vice versa. The schemata may be simple, they are unfortunately also wrong, because they are a bit too optimistic: they don't exclude that both machines simultaneously enter their respective critical sections. If both machines are outside their critical section — say somewhere in the block left blank — then both LA and LB **false**. If now simultaneously they enter in their upper block, they both find that the other machine does not impose any obstacle in their way, and they both go on and arrive simultaneously in their critical section.

2.2 Pessimistic attempt

So we have been too optimistic. The error has been that the one machine did not know, whether the other was already inquiring about his state of progress. The schemata of Fig. 2 make a much more reliable impression, and it is easy to verify that they are totally secure. e.g. machine A can only start to execute section TA, after, while LA is being **true**, it has been verified that LB is false. No matter how soon after the knowledge of this fact to machine A becomes obsolete, because machine B in its upper block performs LB $\leftarrow$ **true**, machine A can safely enter section TA, because machine B will be neatly held up until after finishing section TA LA is being changed to **false**. This solution is perfectly safe, but we should not think that with this we have solved the problem, because this solution is too safe, i.e. there is a chance that the whole teamwork of these machines halts. If they run the upper block in their schema simultaneously, then both machines run in the subsequent wait and continue into eternity of days politely facing the same door, saying, "After you", "After you".

2.3 Dekker's solution

We have been too pessimistic. You see, the problem is not trivial. After these two try-outs, it was not at all obvious, to me at least, that there was a safe solution, that did not also contain the possibility of a dead end. I have passed the problem in this form to my former colleagues of the Computational Department of the Mathematical Centre, adding that I did not know if it has a solution. Dr. T. J. Dekker has the honour of being



(a) Machine A

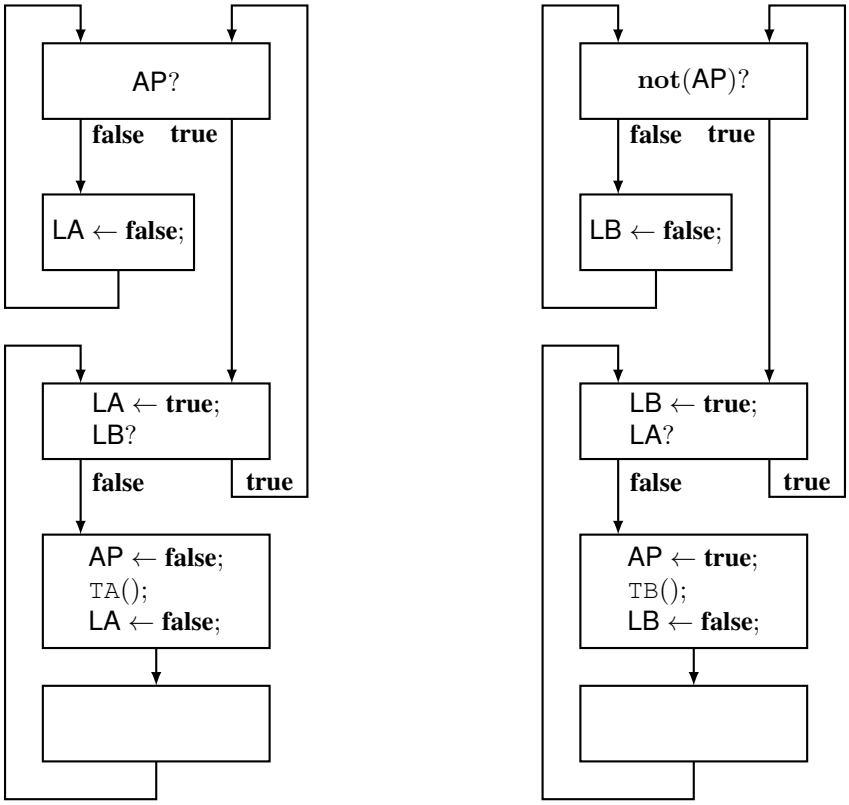
(b) Machine B

Figure 2: Second attempt to solve mutual exclusion

the first to have found a solution, which was also symmetrical in both machines. This solution is shown in Figure 3. A third logical variable is being introduced, namely AP, which means that in case of doubt machine A has priority. Due to the asymmetric significance of this logical variable, portions of these programs more or less mirror each other, but functionally the solution is, however, completely symmetrical and no one machine has a precedence over the other. To prove that this solution conforms to the stated requirements, we first observe that the sections `TA` and `TB` are preceded by the lock, which we have already identified in Figure 2 as safe. Simultaneous execution is thus impossible. We only need to verify that it is also impossible that both machines keep waiting before their critical section. In these wait cycles, the value of AP will not be changed. If we restrict ourselves to the case that AP is **true**, then machine A can only continue cycling if also LB is **true**; under the condition AP **true** machine B though can only remain running in the cycle, that sets LB to **false**, i.e. that forces machine A out of its cycle. In the case of the AP **false** the analysis follows in the same way. As AP as a logical variable is either **true** or **false**, thus the possibility of eternal waiting for each other is also excluded.

Q.E.D.

Dekker's solution dates back to 1959 and for almost three years, this solution has been considered a "curiosity", until these issues at the beginning of 1962 suddenly became relevant for me again and Dekker's solution acted as the basis of my next attempts.



(a) Machine A

(b) Machine B

Figure 3: Dekker's solution

3 Introduction of the semaphores

3.1 Spin lock algorithm

The problem had by now changed drastically. In 1959, the question was of whether the described capability for communication between two machines made it possible to couple two machines, such that the execution of the critical sections was mutually excluded in time. In 1962, a much wider group of issues was considered and the question was also changed to "What communication possibilities between machines are required, to play this kind of games as gracefully as possible?"

Dekker's solution was used as a starting point in different ways. As it is unclear in this solution, what one should do, when the number of machines increases, and the challenge remains to ensure that at any time there is only one critical section treated, it was an incentive to find other ways. By analysing the difficulties that Dekker had to overcome, we could get a clue what would be a more hopeful path to follow.

The difficulties arise because if one of the machines inquired after the value of a common logical variable, this information can immediately afterwards already be obsolete, even before the informing machine has effectively acted upon it. In particular: before the informing machine has had the possibility to inform its

partners that he "has taken a snapshot" of the value of a shared variable. The hint, which can be extracted from this reflection is that one should look for more powerful "elementary" operations on the common variables, in particular for operations, in which the limit of one-way information transmission between the machine and shared memory no longer applies.

The most promising operation we could think of, is the operation `falsify`. This asks after the value of a logical variable and the fact that this operation is performed is revealed in half of the cases. The operation falsify exits, forcing the logical variable, which it operated on, with value **false**. The operation `falsify` can be defined as follows: [1]

boolean `falsify`(**boolean reference** S) **is**
    boolean Q ← S;
    S ← **false**;
    return Q;
end

provided we hereby state that the operation `falsify`, in spite of its sequential definition should be considered as an elementary instruction of the repertoire of machines. Where "elementary" in this case means that during the execution of `falsify` by one of the machines the logical variable in question is considered inaccessible to all other machines. [2]

That this operation is a step in the right direction, follows from the fact that we immediately have the solution for a cloud of machines $X_1, X_2, X_3, \ldots$, each with its critical path $\texttt{TX}_i$, that all have to exclude each other in time. We can play this with one common logical variable, say SX indicating that none of the machines is executing its critical path. The programs now all show the same structure: [3]

while true do
    while ***not*** `falsify`(SX) **do** ;
    $\texttt{TX}_i$() ; `// Critical section`
    SX ← **true**;
    process X_i ; `// Non critical section`
end

Note, that in the programs for the separate machines X_i nowhere is it stated, how many machines X_i actually exist: we can by merely adding, expand the cloud of machines X_i.

3.2 Definition of a semaphore

The programmed wait cycle that exists herein, is of course very nice, but it did little to what our goal. A tiny wait cycle is indeed the way to keep a machine busy "without effect". However, if we consider now, that will be the majority of these machines will be simulated by a central computer so that any action in one machine can only be performed at the expense of the effective speed of the other machines, then it is very costly using a wait cycle to demand attention of the central computer for something completely useless. As long as the wait cycle cannot be exited, in our opinion the speed of this machine may be reduced to zero. To express this we introduce a statement instead of wait cycle, a basic instruction in the repertoire of the machines, that may

[1] Original text:
boolean procedure `falsify` (S); **boolean** S;
begin boolean Q; `falsify`:= Q:= S; **if**(Q) **then** S:= **false**; **end**

[2] The operation `falsify` is now usually identified as `testAndSet`(**false**).

[3] Original text: LX_i: **if** non `falsify`(SX) **then goto** LX_i; $\texttt{TX}_i$(); SX:= **true**; **process** X_i; **goto** LX_i

take a very long time. We indicate this with a P (from *Passering*, in English: to Pass); in anticipation of future needs, we represent the statement SX ← **true** by V(SX) — with V of *Vrijgave* (in English: to Release) (This terminology is taken from the railway environment. In an earlier stage the common logical variables were called "Semaphores" and if their name starts with an S, it is a reminiscence of it). The text of the programs in this new notation is as follows: [4]

while true do
  P(SX);
  $TX_i()$; // Critical section
  V(SX);
  process X_i ; // Non critical section
end

The operation P is a tentative pass. It is an operation, which can only be terminated at a time, when the associated logical variable has the value **true**, termination of the operation P also implies, that the value of the associated logical variable is again set to **false**. Operations P can be carried out simultaneously, the termination of a P operation however is considered as an indivisible event.

The introduction of the operation V is more than introducing a shorthand notation. Because, when we still had the wait cycle using the operation falsify, the individual machines had the duty to detect the equalization to **true** of the common logical variables, which were waiting for the event and in that sense waiting was a fairly active business for a relatively neutral event. With the introduction of the P operation as we more or less achieved, that we say to a machine, which temporarily has to stop, "Go to sleep, we will wake you up when you can progress." But this means that assigning **true** to a common logical variable, for which there might be a machine waiting, no longer is a neutral event like any other value assignment. Here namely we might have to attach the side effect, that one of the "dormant" machines should be "woken". It means that the logical variables, which while being **false** might be blocking one or more machines, need not be permanently monitored for, but only when the transition to the value **true** will be signalled to a "central alarm-clock". If the "central alarm-clock" keeps record for each common logical variable, which machines are waiting in this moment for each logical variable, the central alarm-clock can, if it is explicitly alerted on becoming **true** of one of thos e common logical variables, decide quickly, whether this transition is a motive to wake up a sleeping machine and if so which one. The operation V can now be considered as a combination of assigning the value **true** to a semaphore, plus alerting the central alarm clock on this event. It is clear that this central alarm-clock can not function, if we allow that the common logical variables may also be changed privately, "secretly", by normal value assignments (of the neutral type S ← **true**). To underline this ban, I call the common logical variables from now on "semaphores" and I demand, that the only way, in which a machine will be able to communicate with an existing semaphore, will be the P operation and the V operation.

3.3 Using semaphores for mutual exclusion

And herewith the operation falsify, which by the attractiveness of its simplicity for our comprehension has been so helpful, has again disappeared from the scene and the question is justified, whether we with the introduction of the semaphore, that is only accessible through the operations P and V, have not thrown the baby out with the bathwater. This fortunately is not the case. By removing the operation falsify from the basic repertoire of the various machines and to replace it by the operations P and V, we have lost nothing: be

[4] Original text: LX_i: P(SX); $TX_i()$; V(SX); **process** X_i; **goto** LX_i

the introduction the operations P and V no empty gesture, there must be at least one semaphore, on which they can operate. This one semaphore — call him SG, the Semaphore-General — is sufficient to add the operation falsify on any common logical variable to the repertoire of all machines. Because, if we define in each machine: [5]

boolean falsify(**boolean reference** GB) **is**
    boolean Q;
    P(SG);
    Q ← GB;
    GB ← **false**;
    V(SG);
    return Q;
end

and we furthermore agree that any reference to a common logical variable will be encapsulated between the statements P(SG) and V(SG) (such that the assignment $\mathsf{GB}_1 \leftarrow$ **true** in each machine will now have the form P(SG); $\mathsf{GB}_1 \leftarrow$ **true**; V(SG), we have ensured that we comply to the secondary condition, that "the logical variable in question during operation falsify by one of the machine is inaccessible for all other machines". We did not shoot ourselves in the foot with the barter.

3.4 Using semaphores for signaling

We now will show that the operations P and V in their scope are not limited to mutual exclusion, but that we can use them to synchronize two machines with each other. Suppose that we have two cyclic machines, say X and Y, each in their cycle has a critical section, say TX respectively. TY, wherein the requirements are as following:

1. the execution of the critical section TX may in time not coincide with that of TY;
2. the execution of the sections TX and TY must be alternating events (to this condition we were referring when we spoke of "mutual synchronization").

If you want to imagine an entourage, in which one is placed before this problem, then think about the following. Process X is a number generating process (a calculation process), process Y is a number consuming process (a typewriter) and these two processes are linked by a "counter" having the capacity of one digit. The action TX is to deposit the next digit to type on an empty counter, the piece TY represents the pick up of the digit from the full counter. The strict alternation arises from the requirement that on one hand none of the offered digits to the counter "may be lost" but has necessarily to be typed, and on the other hand — if for some time there is no offer is — the typewriter should not go about filling his time by yet again type the last digit.

We can accomplish this with two semaphores, say SX and SY, where: SX now means that first it is section TX's turn, and SY means that first it is section TY's turn. For machines X and Y programs now read: [6]

[5] Original text:
boolean procedure falsify (GB); **boolean** GB;
begin boolean Q; P (SG) falsify:= Q:= GB; **if**(Q) **then** GB:= **false**; V (SG) **end**

[6] Original text: LX: P(SX); TX(); V(SY); **proces** X; **goto** LX and LY: P(SY); TY(); V(SX); **proces** Y; **goto** LY.

while true do
| P(SX);
| TX();
| V(SY);
| **proces** X;
end

while true do
| P(SY);
| TY();
| V(SX);
| **proces** Y;
end

We can immediately extend this in two different ways. If we exchange the machine X by a cloud of machines X_i, who all want to use the same counter to discharge their production and exchange the machine Y replaced by a cloud machines, all of which are willing to take of the information off the counter (regardless by which the machines X_i it was placed there, therefore the image of typewriters is no longer suitable), then the programs for machines X_i and Y_j read in analogy to the above-programs: [7]

while true do
| P(SX);
| $\mathtt{TX}_i()$;
| V(SY);
| **proces** X_i;
end

while true do
| P(SY);
| $\mathtt{TY}_i()$;
| V(SX);
| **proces** Y_i;
end

If we had also had a group of machines Zk, and had the problem been, that the implementation of a TX-section, a TY-section and a TZ-section should follow each other cyclical in that order, then the programs read: [8]

while true do
| P(SX);
| $\mathtt{TX}_i()$;
| V(SY);
| **proces** X_i;
end

while true do
| P(SY);
| $\mathtt{TY}_i()$;
| V(SZ);
| **proces** Y_i;
end

while true do
| P(SZ);
| $\mathtt{TZ}_k()$;
| V(SX);
| **proces** Z_k;
end

We note here, that in the last two examples the machines X_i are equal: they represent only that the process TX has to be inter-spaced by something, they leave the question open, how complex these intermediate operations will be realized.

The next task may prove that our operations P and V in their current form are not quite powerful enough. We consider therefore the following configuration. We have two machines A and B, each with their critical section TA respectively TB; these sections are completely independent of each other, but there is a third player, machine C, with its critical section TC and the execution of TC may in time not coincide with that of TA, nor with that of TB. We can do this with two semaphores say: SA and SB. For machines A and B, the program is simple, namely [9]

[7] Original text:
LX_i: P(SX); $\mathtt{TX}_i()$; V(SY); **proces** X_i; **goto** LX_i and
LY_j: P(SY); $\mathtt{TY}_j()$; V(SX); **proces** Y_j; **goto** LY_j.

[8] Original text:
LX_i: P(SX); $\mathtt{TX}_i()$; V(SY); **proces** X_i; **goto** LX_i and
LY_j: P(SY); $\mathtt{TY}_j()$; V(SZ); **proces** Y_j; **goto** LY_j and
LZ_k: P(SZ); $\mathtt{TZ}_k()$; V(SX); **proces** Z_k; **goto** LZ_k.

[9] Original text:
LA: P(SA); TA(); V(SA); **proces** A; **goto** LA and
LB: P(SB); TB(); V(SB); **proces** B; **goto** LB.

```
while true do                    while true do
    P(SA);                           P(SB);
    TA();                            TB();
    V(SA);                           V(SB);
    proces A;                        proces B;
end                              end
```

but for machine C the next program is although safe, unacceptable: [10]

```
while true do
    P(SA);
    P(SB);
    TB();
    V(SB);
    V(SA);
    proces B;
end
```

In fact: machine C might have successfully completed the statement P (SA) at a time, that machine B is busy in section TB. As long section TB is not yet completed, machine C can not continue — that is correct — but also machine A can no longer execute his section TA and that is not correct: indirectly, via C machine here the sections TA and TB are yet again coupled. The operation P (SA) of machine C hastily put the semaphore SA to false, thereby hindering A machine unnecessarily.

We will therefore extend the operations P and V to operations, to which we can offer an arbitrary number of arguments — exceeding the limits of ALGOL 60, but that does matter very little in this context. For the V operation the meaning is clear: it is no different than the simultaneous assignment, which assigns the value **true** to all semaphores passed. The P operation with a number of arguments, e.g. $\mathtt{P}(S_1, S_2, S_3)$, is an operation which can terminate only at the moment, that all semaphores passed have the value **true** (in our example therefore the moment when is valid: $S_1 \wedge S_2 \wedge S_3$, termination of a P operation in this context implies that simultaneously a value of **false** is assigned to all semaphores. And after this expansion the termination of a P operation is again considered an indivisible event.

With this extension, we are able to define machine C, such that the signalled objection no longer presents itself: [11]

```
while true do
    P(SA, SB);
    TB();
    V(SA, SB);
    V(SA);
    proces B;
end
```

[10] Original text: LC: P(SA); P(SB); TC(); V(SB); V(SA); **proces** C; **goto** LC.
[11] Original text: LC: P(SA, SB); TC(); V(SA, SB); **proces** C; **goto** LC.

4 A concluding example

Finally I wish to show, although this is certainly not the application for which the P and V operations are designed, how this will allow us to calculate the dot product of two vectors, without committing us to the order in which the products of the elements are added together.

Suppose that we wish to calculate the sum $\sum_{i=1}^{5} A[i] * B[i]$; Suppose that have been introduced the 7 semaphores: Ssom, Sklaar and five semaphores Sterm$[i]$ with $i \in \{1, 2, 3, 4, 5\}$; all 7 with the initial value **false**;

Suppose, that already have been created five machines with the structure ($i \in \{1, 2, 3, 4, 5\}$): [12]

while true do
  P (Ssom, Sterm [i]);
  scapro $\leftarrow$ scapro $+ A[i] * B[i]$;
  $n \leftarrow n - 1$;
  if $n = 0$ **then** V (Sklaar) **else** V (Ssom);
end

all executing their (only) P operation;

Consider now a sixth machine, that under these conditions will begin the following statements:

...;
$n \leftarrow 5$;
scapro $\leftarrow 0$;
V(Ssom, Sterm$[1]$, Sterm$[2]$, Sterm$[3]$, Sterm$[4]$, Sterm$[5]$);
// program, that does not modify names semaphores, scalars and
   vector elements
P(Sklaar);
...

After the last P operation of the sixth machine is finished, n has the value 0, scapro the value of the requested sum and all semaphores and five former machines are returned in their original condition.

References

[1] Edsger W. Dijkstra. *Over de sequentialiteit van procesbeschrijvingen.* (1962-1963)

[2] Martien van der Burgt, Heather Lawrence. www.cs.utexas.edu/users/EWD/translations/EWD35-English.html

[12] Original text:
$Lterm[i]$: P (Ssom, Sterm [i]);
  scapro := scapro $+ A[i] * B[i]$;
  $n := n - 1$;
  if $n = 0$ **then** V (Sklaar) **else** V (Ssom);
  goto $Lterm[i]$

Over de sequentialiteit van procesbeschrijvingen.

Het is niet ongebruikelijk, wanneer een spreker zijn voordracht begint met een inleiding. Omdat ik mij hier mogelijk richt tot een gehoor, dat als geheel minder vertrouwd is met de problematiek, die ik wil aansnijden en de terminologie, die ik zal moeten gebruiken, wilde ik in dit geval ter introductie graag twee inleidingen houden, nl. een om de achtergrond van de problemen te schilderen en een tweede, om U een gevoel te geven voor de aard van de logische problemen, die wij tegen het lijf zullen lopen.

Mijn eerste inleiding is een historische. Eerst moet ik U vertellen, wat U zich moet voorstellen bij een sequentiele procesbeschrijving. U kent allen dergelijke procesbeschrijvingen. Beschouw bv. de beschrijving van de constructie van de hoogtelijn uit het hoekpunt C van de driehoek ABC. Deze kan als volgt luiden:

1) trek de cirkel met middelpunt A en straal = AC;
2) trek de cirkel met middelpunt B en straal = BC;
3) trek de rechte, die bepaald is door de snijpunten van de onder 1) en 2) genoemde cirkels.

Dit is een beschrijving ten bate van iemand, wiens "parate kennis" de constructie van de hoogtelijn niet omvat; ten bate van hem is de constructie van de hoogtelijn opgebouwd uit een aantal handelingen van een beperkter repertoire, handelingen, die hij in de opgegeven volgorde, de een na de ander, dwz. sequentieel, kan uitvoeren. Het zal het oplettende lezertje niet ontgaan zijn, dat wij hierbij meer gespecificeerd hebben, dan strikt noodzakelijk: het is wel essentieel, dat handeling 3) pas uitgevoerd wordt, nadat handelingen 1) en 2) voltooid zijn, handelingen 1) en 2) hadden best in de omgekeerde volgorde plaats mogen vinden, sterker, een tekenaar, die twee passers tegelijkertijd bedienen kan, bij wijze van spreken met elke hand één, mag handelingen 1) en 2) simultaan uitvoeren. Maar deze vrijheid is in onze procesbeschrijving geheel onder tafel geraakt.

Nu een numerieker voorbeeld, dat iets meer aansluit bij de denkwereld, waarin de straks te behandelen problemen actueel zijn geworden. Gegeven de waarden van de variabelen a, b, c en d, gevraagd te berekenen de waarde van de variabele x, bepaald door de formule "x:=(a+b)*(c+d)". Wanneer het uitvoeren van deze waardetoekenning niet tot het primitieve repertoire van de rekenaar -en denkt U nu langzamerhand maar aan de rekenmachine- behoort, dan dient dit rekenvoorschrift uit elementairdere stappen opgebouwd te worden, bv.:

1) bereken t1:= a + b;

2) bereken t2:= c + d;

3) bereken x:= t1 * t2.

Dit rekenvoorschrift bevat precies dezelfde overbepaaldheid als onze hoogtelijn-constructie: de volgorde van de eerste twee stappen doet niet ter zake, zij zouden zelfs simultaan uitgevoerd kunnen worden.

De bovenstaande procesbeschrijving is representatief voor de werkingswijze van meer klassieke rekenautomaten. Het repertoire van elementaire operaties is, omdat voor elke operatie speciale technische voorzieningen vereist zijn, eindig en om financiele redenen zelfs heel beperkt; deze machines danken hun enorme flexibiliteit aan het feit, dat willekeurig ingewikkelde algebraische expressies als boven geillustreerd uitgewerkt kunnen worden door een sequens van dergelijke elementaire operaties. Inhaerent aan deze sequentiele beschrijving van het proces is, dat men meer specificeert, dan strikt noodzakelijk. Dit heeft lange tijd niet gehinderd, maar in de laatste jaren is daar verandering in gekomen, en wel door twee oorzaken: verandering van de structuur van machines en de komst van een nieuwe klasse problemen, waarvoor men deze machines zou willen inzetten.

De klassieke sequentiele machine doet "een ding tegelijk" en verricht de ene functie na de andere. Om der wille van de efficiency is men voor specifieke functies meer specifieke apparatuur gaan bouwen, maar als je er dan aan vast houdt, dat de verschillende operaties strict sequentieel uitgevoerd zullen worden, dan dringt zich van een machine plotseling het beeld op van een samenspel van N organen, waarvan er op elk discreet ogenblik maar 1 of 2 werken. Als N groot is, betekent dit, dat het merendeel van je machine het merendeel van de tijd stilstaat en dat is kennelijk niet de bedoeling.

De tweede oorzaak is de komst van een nieuwe categorie van problemen; oorspronkelijk werden rekenautomaten alleen gebruikt voor van te voren volledig gedefinieerde processen, en de machine kon het proces vervolgens op zijn eigen houtje, in zijn eigen tempo uitvoeren. Zodra men echter het informatieverwerkende proces samen wil laten spelen met een of ander ander proces -bv. een chemisch proces, dat door de rekenautomaat gestuurd moet worden- dan treden er heel andere problemen op. Aanhakend bij ons voorbeeld van de berekening van x:=(a + b)*(c + d) zijn we dan niet meer in de omstandigheid, dat het er niet meer toe doet, welke som het eerste gevormd wordt. Als de gegevens a,b,c en d op onbekende momenten door de omgeving van de rekenautomaat verschaft worden en het de taak van de machine is, om de waarde van x zo snel mogelijk af te leveren, dan betekent dit, dat de auto-

maat een deelberekening maar vast uit moet voeren, zodra de daarvoor vereiste gegevens binnen zijn: welke optelling men dan eerst uitgevoerd wil hebben, zal afhangen van de volgorde. waarin de gegevens a, b, c en d ter beschikking komen. Dit was mijn eerste inleiding en ik hoop U hiermee een idee gegeven te hebben, van wat U zich bij een sequentiele procesbeschrijving voor moet stellen, en waarom de vraag gesteld is naar minder strict sequentiele prcesbeschrijvingen.

Nu mijn tweede inleiding, die bedoeld is om U van meet af aan enig idee te geven van de logische problemen, die we door het verlaten van de stricte sequentialiteit ons op de hals zullen halen.

Evenals in een normaal stuk algebra variabelen met letters, algemeen "namen", aangeduid worden, heeft men in het proces van informatieverwerking aanhoudend de plicht tot identificatie, de plicht om bij de naam het hierdoor benoemde object te vinden. Onder omstandigheden zou men graag willen beschikken over het selectieproces, waarover de juffrouw in de klas beschikt, wanneer zij zegt "Jantje, kom eens voor het bord." Aangenomen, dat de klas als geheel oplet en gehoorzaam is, werkt dit proces feilloos onder de aanname, dat er 1 en slechts 1 Jantje in de klas zit. Het bijzondere van dit selectiemechanisme is, dat het werkt ongeacht het aantal kinderen in de klas. Deze idylle is helaas technisch niet of hoogstens gebrekkig te verwezenlijken, en en wij krijgen een probleemstelling, die meer in overeenstemming is met de werkelijkheid, wanneer de kinderen niet spontaan op het noemen van hun naam reageren en de juffrouw, wanneer zij Jantje voor het bord wil hebben, genoodzaakt is om Jantje bij het oor te vatten en daaraan voor het bord te slepen. Als dit nu een ouderwetse school is, waarin ieder kind zijn vaste plaats in de klas heeft, dan kan de juffrouw, als zij aan het begin van het jaar de kinderen op hun plaats gezet heeft, ze voor haar eigen gemak herdopen, en de naam "Jantje" -die voor het vinden van het jongetje niet erg behulpzaam is- vervangen door een identificatie, die blijkens zijn structuur meteen definieert, welk oor ze vatten moet: ze kan het jongetje noemen "derde rij, tweede bank". Dit is in wezen de techniek, die in klassieke automaten bij klassieke toepassingen gevolgd wordt. In modernere toepassingen kan men dit niet meer doen, en moet men bv. opgewassen zijn tegen de problemen, die de juffrouw krijgt, als de kinderen kunnen kiezen, waar ze gaan zitten.

Om het de juffrouw niet onmogelijk te maken, nemen we aan dat elk der onwillige kinderen een naamplaatje opgespeld heeft. Als de juffrouw nu Jantje voor het bord wil hebben, moet zij dus Jantje opzoeken. De juffrouw werkt sequentieel, kan

slechts een naamplaatje tegelijkertijd lezen, en de juffrouw werkt systematisch: zij heeft voor zichzelf de banken op een of andere manier uniek geordend en als zij Jantje wil hebben, dan loopt zij de banken in deze volgorde af, zich steeds afvragend "Ben jij Jantje?", zo nee, dan gaat ze naar de volgende bank. Dit proces werkt feilloos, mits wij kunnen garanderen, dat de kinderen tijdens de rondgang van de juffrouw niet gaan verzitten, mits wij kunnen garanderen, dat de twee processen "kindlocalisatie" en "kinderpermutatie" niet simultaan uitgevoerd worden. Want U begrijpt ook wel, dat Jantje, zodra hij door heeft, dat hij de gezochte figuur is, achter in de klas gaat zitten en zodra de juffrouw de voorste helft van de klas doorzocht heeft, springt hij gauw naar voren. Wij kunnen de juffrouw verfijnen en haar, als ze de hele klas doorzocht heeft en Jantje niet gevonden heeft met frisse moed van voren af aan laten beginnen, maar dit is geen oplossing, want Jantje springt dan gauw weer naar achteren. Nu zou U kunnen opmerken, dat als die permutaties nu niet met zoveel kwaadwilligheid, maar slechts random uitgevoerd werden, dat dan de verwachtingstijd voor het localiseren van Jantje eindig blijft, ja zelfs door zijn vrijheid rond te springen niet groter is geworden. Deze opmerking, hoe hoopvol, brengt geen soulaas. Zij die zich verdiept hebben in de "Theory of computability" weten hoeveel gewicht gehecht wordt aan een eindige, dwz. een gegarandeerd eindigende algorithme, en zij zullen begrijpen, dat een proces, dat potentieel oneindig lang doortolt, geen aantrekkelijk element is. Veel erger is, dat het in de praktijk veelal de vraag is, of wij de permutaties van de kinderen wel als random mogen beschouwen. Als de kans eindig is, dat als de juffrouw een keer door de klas geweest is, de kinderen weer precies zo zitten als toen de juffrouw begon te zoeken, dan kon dit best eens de eerste periode van een zuiver periodiek verschijnsel zijn.

Maar zelfs als wij het risico, dat we Jantje nooit zullen vinden voor lief nemen, dan nog is haar ellende niet te overzien, want naast het risico, dat ze Jantje niet vindt, loopt ze het veel grotere gevaar, dat ze het verkeerde kind voor het bord sleurt, nl. als de kinderen tussen het moment, dat zij voldaan geconstateert heeft "Zo, jij ben dus Jantje" en het moment, dat zij, op grond van deze constatering haar hand naar het bijbehorend oor heeft uitgestoken, nog gauw van plaats verwisselen. In een informatieverwerkend proces zou dit neerkomen op een ramp. En dit is het einde van mijn tweede inleiding, waarvan ik hoop, dat zij U enig idee gegeven heeft van het soort probleem, dat wij tegen zullen komen.

Aan het einde van mijn eerste inleiding heb ik twee oorzaken genoemd, die voor een groeiende belangstelling in minder strict sequentiele procesbeschrijvingen

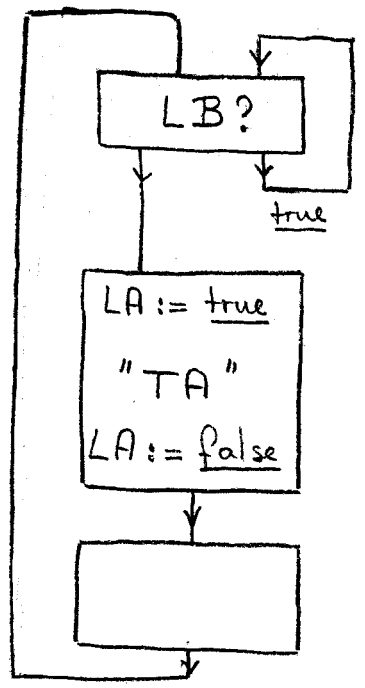
verantwoordelijk waren. Ik heb ze slechts luchtig aangestipt en het is U misschien niet opgevallen, dat ze aan bijna tegengestelde entourages ontleend waren. In het eerste geval noemde ik de toenemende complexiteit van machines, die nu opgebouwd zijn uit een aantal min of meer autonoom werkende onderdelen en de opgave was, om deze samen één proces uit te laten voeren. In het tweede geval, waar we de automaat in een z.g. "real time application" bekeken, werd het beeld opgeroepen van één enkel apparaat, dat de mogelijkheid had bij een verandering van de externe urgentie-situatie over te schakelen op het nu urgenste van zijn mogelijke taken. Deze twee arrangementen, een samenspel van een aantal machines aan één proces en omgekeerd, één machine zijn aandacht verdelend over een aantal processen, zijn lange tijd als elkaar wezensvreemd beschouwd. Men heeft er zelfs verschillende namen voor uitgevonden, het ene noemt men "parallel programming" en het andere noemt men "multiprogramming" maar U moet mij niet meer vragen wat wat, want dat kan ik niet meer onthouden, sinds ik ontdekt heb dat de logische problemen, die zij door de non-sequentialiteit van de procesdefinitie oproepen, in beide gevallen exact dezelfde zijn.

Ik zal mij in het volgende bedienen van het beeld van een aantal, onderling zwak gekoppelde, opzichzelf sequentiele machines, daarbij aansluitend op de eerste entourage. Maar dit is slechts een beschrijvingswijze, want met wat ik in het vervolg een machine zal noemen komt niet noodzakelijkerwijze een discreet aanwijsbaar stuk apparatuur overeen: in het volgende zijn mijn onderscheiden machines mogelijk heel abstract en representeren zij niet meer dan een opzichzelf sequentieel deelproces. Dit komt onder andere daarin tot uiting, dat ik een grote voorliefde zal tonen voor een samenspel van N machines, N willekeurig. Als ik de term machine alleen maar bezigde voor een aanwijsbaar functioneel stuk apparatuur, dan zou ik het aantal machines voor elke installatie als vrij constant mogen beschouwen. Nu echter een machine , als personifiering van een opzichzelf sequentiele deeltaak, door de gebruiker ad libitum gecreeerd kan worden, is een dergelijke vaste bovengrens niet meer acceptabel. Ik ga dus praten over het samenspel van N machines, N niet alleen willekeurig, maar als het moet zelfs tijdens het proces variabel. Overtollige machines kunnen vernietigd worden, nieuwe machines kunnen naar behoefte er bij gecreeerd worden en in het samenspel der overigen worden opgenomen.

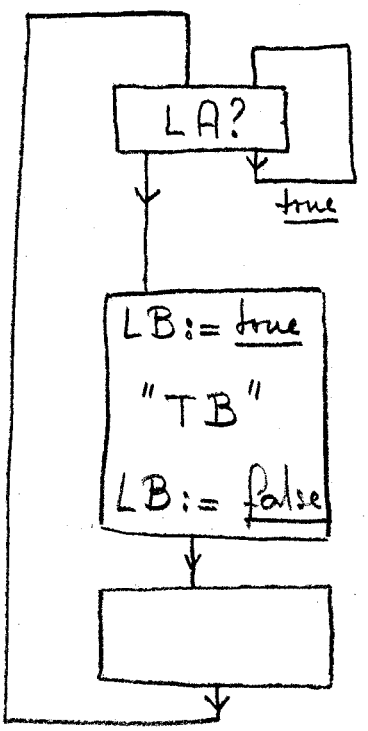
Ik hoop, dat U thans op mijn gezag wilt aannemen, danwel na afloop zelf inziet, dat wij ons zonder de algemeenheid te schaden kunnen beperken tot sequentiele machines, die een of ander cyclisch proces uitvoeren.

Laat ons beginnen met een heel eenvoudig probleem. Gegeven twee machines A en B, beide bezig met een cyclisch proces. In de cyclus van machine A komt een zeker critisch traject voor, TA genaamd, en in die van machine B komt een critisch traject TB voor. De opgave is om er voor te zorgen, dat nooit gelijktijdig de beide machines elk aan hun critische traject bezig zijn. (In termen van onze tweede inleiding: machine A zou kunnen zijn de juffrouw, traject TA het selectieproces van een nieuwe leerling, die voor het bord gehaald moet worden, terwijl het traject TB het proces "kinderpermutatie" representeert.) Er mogen geen veronderstellingen gemaakt worden over de relatieve snelheden der machines A en B, de snelheid, waarmee zij werken hoeft zelfs niet constant te zijn. Het is duidelijk, dat wij de wederzijdse uitsluiting van het critische traject slechts kunnen realiseren, als de twee machines op een of andere manier met elkaar kunnen communiceren. Voor deze communicatie stellen wij enig gemeenschappelijk geheugen ter beschikking, dwz. een aantal variabelen, die voor beide machines in beide richtingen toegankelijk zijn, dwz. waaraan beide machines een waarde kunnen toekennen en waarvan beide machines naar de heersende waarde kunnen informeren. Deze twee handelingen, toekennen van een nieuwe waarde en informeren naar de heersende waarde gelden als ondeelbare handelingen, dwz. als beide machines "tegelijkertijd" een waarde aan een gemeenschappelijke variabele willen toekennen, dan is de waarde na afloop of de ene, of de andere toegekende waarde, maar niet een of ander mengsel. Evenzo: als de ene machine informeert naar de waarde van een gemeenschappelijke variabele op het moment, dat de andere machine er een nieuwe waarde aan toekent, dan dringt tot de vragende machine of de oude, of de nieuwe, maar niet een wilde waarde door. Het zal blijken, dat we ons voor dit doel kunnen beperken tot gemeenschappelijke logische variabelen, dwz. variabelen met slechts twee mogelijke waarden, die we in aansluiting op standaard technieken met "true" respectievelijk "false" aangeven. Voorts nemen wij aan, dat beide machines slechts naar 1 gemeenschappelijke variabele tegelijkertijd kunnen refereren, en dan ook alleen of om een nieuwe waarde toe te kennen of om naar de heersende waarde te informeren. In figuur 1 is in blokschemavorm een tentatieve oplossing gegeven voor de structuur van de programma's voor beide machines. Elk blok heeft zijn ingang boven en zijn uitgang onder. Bij een blok met een dubbele uitgang wordt de keuze bepaald door de waarde van de zojuist aangevraagde logische variabele: heeft deze de waarde true, dan is de uitgang rechtsonder bedoeld, heeft deze de waarde false, dan kiezen we de uitgang linksonder.

Er komen in dit schema twee gemeenschappelijke logische variabelen voor, LA en LB. LA betekent: machine A is in zijn critische sectie, LB betekent: machine B

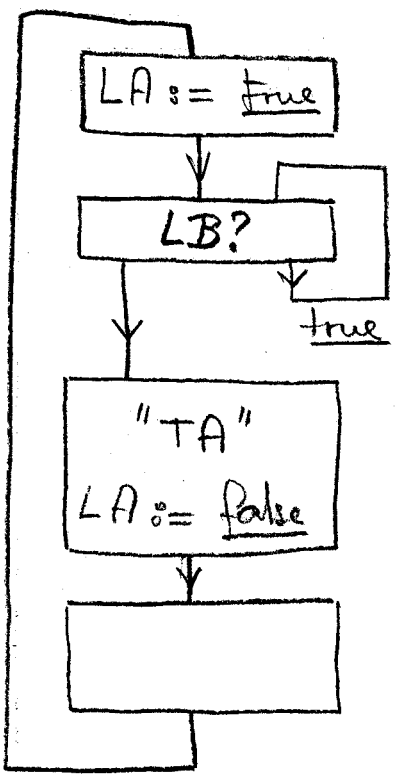


machine A



machine B

Fig 1



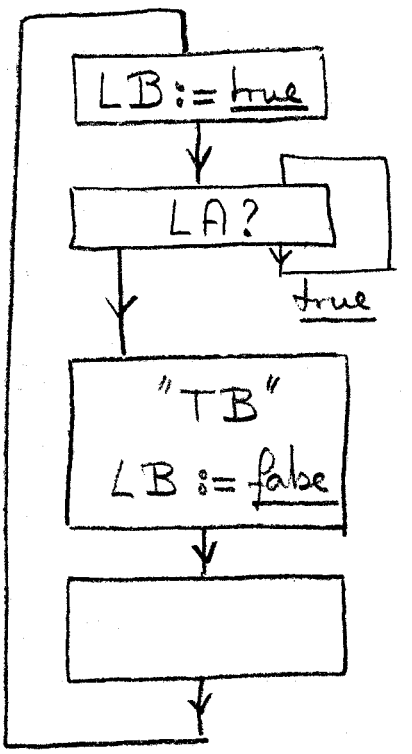


Fig 2.

is in zijn critische sectie. De schema's in fig 1 zijn duidelijk. In het blok bovenaan wacht machine A indien hij daar aankomt op een ogenblik, dat machine B in sectie TB bezig is, totdat machine B zijn critische traject heeft verlaten, wat door de toekenning "LB:= false" gemarkeerd wordt, waardoor dan de wacht van machine A wordt opgeheven. En omgekeerd. De schema's mogen dan eenvoudig zijn, ze zijn helaas ook fout, want ze zijn wat te optimistisch: ze sluiten nl. niet uit, dat beide machines tegelijkertijd in hun respectievelijke critische trajecten terecht komen. Als beide machines buiten hun critische traject zijn -zeg ergens in het blanco gelaten blok- dan zijn zowel LA als LB false. Komen ze nu tegelijkertijd in hun bovenste blok, dan vinden ze beide, dat de andere machine hen geen strobreed in de weg legt, en ze gaan beide door en komen tegelijkertijd in hun critische sectie.

We zijn dus wat te optimistisch geweest. De fout is geweest, dat de ene machine niet wist, of de ander al naar zijn staat van vordering informeerde. De schema's van fig.2 maken een veel betrouwbaardere indruk en het is gemakkelijk te verifieren, dat zij volslagen veilig zijn. Bv. machine A kan slechts aan traject TA beginnen, nadat tijdens het true-zijn van LA geconstateerd is, dat LB false is, en ongeacht hoe snel na deze kennis van machine A dit feit weer obsoleet wordt, doordat machine B gauw zijn bovenste blok "LB:= true" uitvoert, machine A kan dan veilig traject TA ingaan, omdat machine B toch netjes opgehouden wordt totdat LA na afloop van traject TA op false gezet wordt. Deze oplossing is dus volkomen veilig, maar we moeten niet denken, dat we hiermee het probleem opgelost hebben, want deze oplossing is te veilig, er bestaat nl. de kans, dat het hele samenspel van deze machines vastloopt. Als zij tegelijkertijd het bovenste blok van hun schema uitvoeren, dan lopen beide machines in de daarop volgende wacht en blijven tot in eeuwigheid van dagen beleefd voor dezelfde deur staan, zeggende "Na U" ,"Na U". We zijn te pessimistisch geweest.

U ziet, dat het probleem niet triviaal is. Het was mij althans, na deze twee probeersels , helemaal niet zonneklaar, dat er een veilige oplossing bestond, die niet tevens de mogelijkheid van een doodlopende weg in zich hield. Ik heb het probleem toen in deze vorm aan mijn toenmalige collegae van de Rekenafdeling van het Mathematisch Centrum voorgelegd, er bij vertellend, dat ik niet wist, of het oplosbaar was. Aan Dr.T.J.Dekker komt de eer toe als eerste een oplossing gevonden te hebben, die bovendien symmetrisch in de beide machines was. Deze oplossing is weergegeven in fig.3. Er is een derde logische variabele ingevoerd, nl. AP, die betekent, dat in geval van twijfel machine A prioriteit heeft. Door de asymmetrische

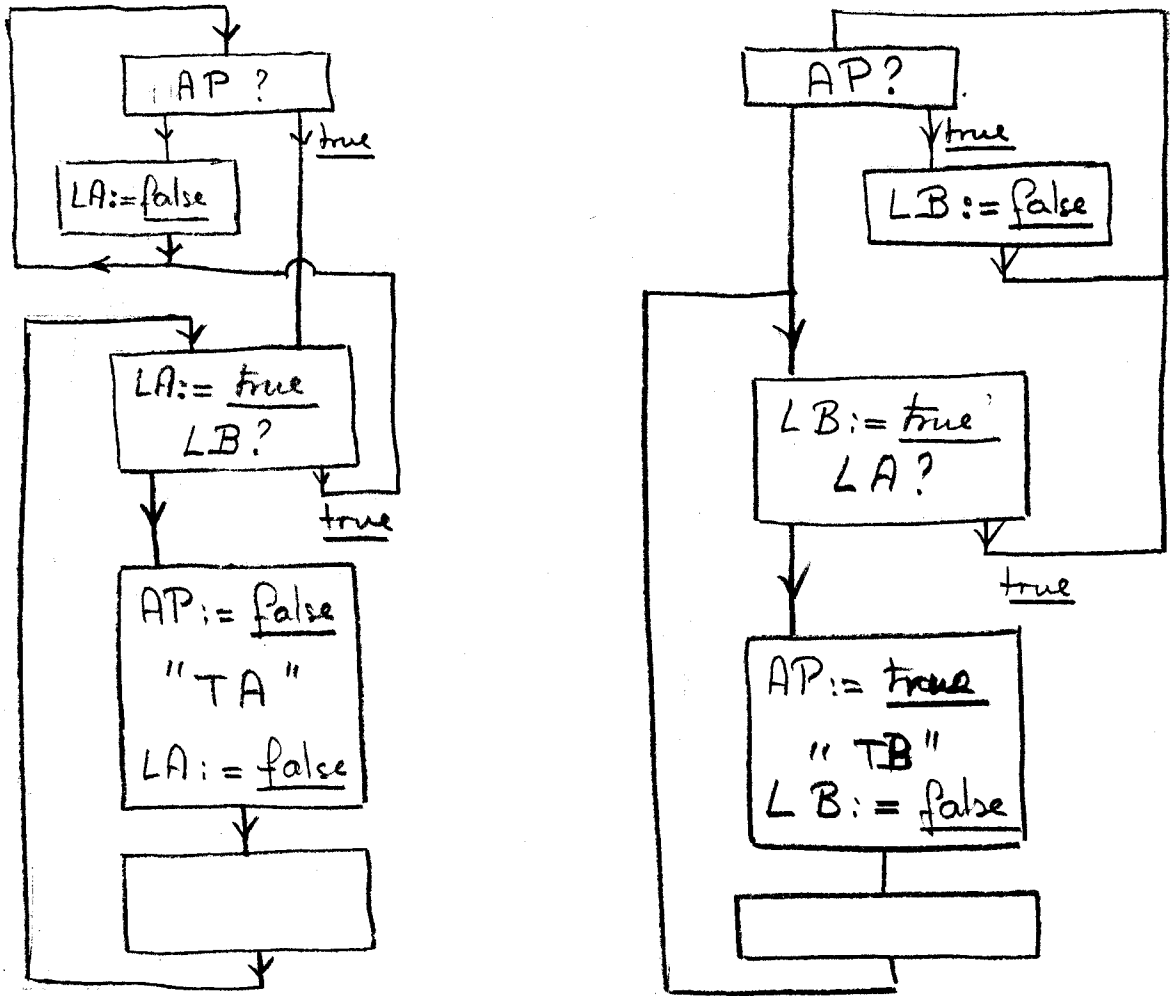


Fig 3.

betekenis van deze logische variabele zijn gedeelten van deze programma's min of meer elkaars spiegelbeeld, functioneel is de oplossing echter helemaal symmetrisch en is niet de ene machine bevoorrecht boven de ander. Om te bewijzen, dat deze oplossing aan de gestelde eisen voldoet, constateren we eerst, dat de trajecten TA en TB voorafgegaan worden door de sluis, die we in figuur 2 al als veilig herkend hebben. Simultane uitvoering is dus onmogelijk. We hoeven slechts te verifieren, dat het nu bovendien uitgesloten is, dat beide machines voor hun critische sectie blijven wachten. In deze wachtcycli wordt de waarde van AP niet gewijzigd. Beperken we ons tot het geval, dat AP true is, dan kan machine A slechts blijven cyclen als ook LB true is; onder de voorwaarde AP true kan machine B echter alleen rond blijven lopen in de cyclus, die LB op false zet, dwz. machine A uit zijn cyclus forceert. In het geval AP false verloppt de analyse op dezelfde manier. Omdat AP als logische variabele of true of false is, is daarmee de mogelijkheid van eeuwig op elkaar blijven wachten eveneens uitgesloten. Q.E.D.

Dekker's oplossing dateert uit 1959 en bijna drie jaar heeft deze oplossing voortgeleefd als "curiositeit", totdat dit soort problematiek aan het begin van 1962 voor mij plotseling weer actueel werd en Dekker's oplossing als uitgangspunt van mijn volgende pogingen gefungeerd heeft.

De probleemstelling was inmiddels wel drastisch veranderd. In 1959 was het de vraag of de beschreven communicatiemogelijkheid tussen twee machines het mogelijk maakte om twee machines zodanig te koppelen, dat de uitvoeringen van de critische trajecten elkaar in de tijd wederzijds uitsloten. In 1962 werd een veel wijdere groep problemen bekeken en werd bovendien de vraagstelling veranderd in "Welke communicatiemogelijkheden tussen machines zijn gewenst, om dit soort spelletjes zo sierlijk mogelijk te spelen?"

Dekker's oplossing heeft op verschillende manieren als uitgangspunt gefungeerd. Omdat het in deze oplossing onduidelijk is, hoe men te werk moet gaan, wanneer het aantal machines uitgebreid wordt, en de opgave blijft om te garanderen, dat er op elk moment hoogstens 1 critische sectie onder behandeling is, was zij een aansporing om naar andere wegen te zoeken. Door een analyse van de moeilijkheden, die Dekker moest overwinnen, konden wij een aanwijzing krijgen wat een hoopvollere weg zou zijn om in te slaan.

De moeilijkheden ontstaan, doordat als een van de machines geinformeerd heeft naar de waarde van een gemeenschappelijke logische variabele, deze informatie onmiddellijk daarna al weer obsoleet kan zijn, nog voordat de informerende machine er effectief op gereageerd heeft, in het bijzonder: nog voordat de informerende machine aan zijn partners heeft kunnen meedelen, dat hij van de waarde van een gemeenschappelijke variabele "een momentopname gemaakt heeft". De hint, die uit deze bespiegeling gehaald kan worden, is dat men moet zoeken naar machtigere "elementaire" operaties op de gemeenschappelijke variabelen, in het bijzonder naar operaties, waarin voor de informatietransport tussen machine en gemeenschappelijk geheugen niet meer de beperking van het éénrichtingverkeer geldt.

De meestbelovende operatie, die we hebben kunnen bedenken, is de operatie "falsify". Deze informeert naar de waarde van een logische en het feit, dat deze operatie is uitgevoerd wordt in de helft van de gevallen kenbaar gemaakt: de operatie falsify laat de logische variabele, waarop hij geopereerd heeft nl. met de waarde "<u>false</u>" achter.

De operatie "falsify" kan als volgt gedefinieerd worden:

"boolean procedure falsify(S); boolean S;
begin boolean Q; falsify:= Q:= S; if Q then S:= false end" ,

mits we hierbij vermelden, dat de operatie falsify, in weerwil van zijn sequentiele definitie, beschouwd moet worden als elementaire opdracht van het repertoire der machines, waarbij "elementair" in dit geval betekent, dat gedurende de operatie falsify door één van de machines de logische variabele in kwestie ontoegankelijk is voor alle andere machines.

Dat deze operatie een stap in de goede richting is, volgt wel uit het feit, dat we nu meteen de oplossing hebben voor een wolkje machines X1, X2, X3,....., elk met zijn critisch traject TXi, die elkaar nu alle in de tijd moeten uitsluiten. We kunnen dit spelen met 1 gemeenschappelijke logische variabele, zeg SX, die aangeeft, dat geen van de machines in zijn critische traject bezig is. De programma's vertonen nu alle dezelfde structuur:

"LXi: if non falsify(SX) then goto LXi; TXi; SX:= true; proces Xi; goto LXi"

waarbij we de aandacht er op vestigen, dat in de programma's voor de aparte machines Xi nergens tot uiting komt, hoeveel machines Xi er eigenlijk zijn: we kunnen dus door loutere toevoeging het wolkje machines Xi uitbreiden.

Het geprogrammeerde wachtcyclusje, dat hierin voorkomt, is natuurlijk heel aardig, maar het beantwoordt wat weinig aan ons doel. Een wachtcyclusje is immers de manier om een machine "zonder effect" bezig te houden. Maar als we nu bedenken, dat het merendeel van deze machines door een centrale computer gesimuleerd zal worden, zodat elke actie in de ene machine slechts plaats kan vinden ten koste van de effectieve snelheid van de andere machines, dan is het wel erg begrotelijk om in een wachtcyclusje aandacht van de centrale computer op te gaan eisen voor iets volslagen nutteloos. Zolang het wachtcyclusje niet verlaten kan worden, mag wat ons betreft de snelheid van deze wachtende machine tot nul zakken, Om dit tot uitdrukking te brengen voeren we in plaats van het wachtcyclusje een statement in, een elementaire opdracht van het repertoire van de machines, die heel heel lang kan duren. We geven dit aan met een P (van Passering); vooruitlopend op latere behoeften representeren we de statement "SX:= true" door "V(SX) -met de V van Vrijgave. (Deze terminologie is ontleend aan het spoorwegwezen: in een eerder stadium heetten de gemeenschappelijke logische variabelen "Seinpalen" en als hun naam met een S begint, dan is dat nog een reminiscentie daaraan.) De text van de programma's luidt in deze nieuwe notatie:

"LXi: P(SX); TXi; V(SX); proces Xi; goto LXi" .

De operatie P is een tentatieve passering. Het is een operatie, die slechts beeindigd kan worden op een moment, dat de er bij opgegeven logische variabele de waarde true heeft, beeindiging van de operatie P houdt tevens in, dat aan de logische variabele in kwestie weer de waarde false wordt toegekend. Operaties P kunnen wel simultaan plaatsvinden, het beeindigen van een P-operatie wordt echter weer beschouwd als een ondeelbare gebeurtenis.

De invoering van de operatie V is meer dan een verkorte schrijfwijze invoeren. Immers: toen wij het wachtcyclusje met behulp van de operatie falsify nog hadden, rustte de detectieplicht ten aanzien van het gelijk true worden van de gemeenschappelijke logische variabelen op de individuele machines, die op deze gebeurtenis stonden te wachten en in die zin was het wachten een tamelijk actieve bezigheid op een betrekkelijk neutrale gebeurtenis. Met de invoering van de P-operatie hebben wij zo ongeveer bereikt, dat wij tegen een machine, die voorlopig niet verder kan, zeggen: "Ga maar slapen, we maken je wel weer wakker, als je verder mag." Maar dit betekent, dat het op true zetten van een gemeenschappelijke logische variabele, waarop misschien een machine staat te wachten, niet meer een neutrale gebeurtenis is als ieder andere waardetoekenning: hier kan nl. het neveneffect aan verbonden moeten worden, dat een van de "slapende" machines "gewekt" moet worden. Het betekent, dat de logische variabelen, die gedurende false-zijn één of meer machines op kunnen houden, niet permanent geobserveerd hoeven te worden, mits bij overgang naar de waarde true een "centrale wekker" op deze overgang attent gemaakt kan worden. Als de centrale wekker per gemeenschappelijke logische variabele een administratie bij houdt, welke machines op elke logische variabele op het ogenblik wachten, dan kan de centrale wekker, mits deze expliciet op het true worden van een dergelijke gemeenschappelijke logische variabele geattendeerd wordt, snel decideren, of er op grond van deze overgang een slapende machine gewekt kan worden en zo ja welke. De operatie V kan nu opgevat worden als een combinatie van het toekennen van de waarde true aan een seinpaal, plus het attenderen van de centrale wekker op deze gebeurtenis. Het si duidelijk, dat deze centrale wekker zijn werk niet kan doen, als we toestaan, dat de gemeenschappelijke logische variabelen ook nog, door normale waardetoekenningen (van het neutrale type "S:= true"), "stiekem", ondershands gewijzigd kunnen worden. Om dit verbod te onderstrepen, noem ik de gemeenschappelijke logische variabelen van nu af aan "seinpalen" en stel ik, dat de enige mogelijkheid, waarop een machine met een bestaande seinpaal zal kunnen communiceren, zal zijn de P-operatie en de V-operatie.

En hiermede is de operatie "falsify", die ons door de aantrekkelijkheid van zijn eenvoud bij de begripsvorming zo behulpzaam is geweest, weer van het toneel verdwenen en de vraag is gerechtvaardigd, of we door de introductie van de seinpaal, die slechts via de operaties P en V toegankelijk is, het kind niet met het badwater hebben weggegooid. Dit is gelukkig niet het geval. Door de operatie "falsify" van het elementaire repertoire van de onderscheiden machines af te voeren en daar de operaties P en V voor in de plaats te geven, hebben wij niets verloren: wil de invoering van de operaties P en V niet een lege geste zijn, dan moet er dus minstens 1 seinpaal zijn, waarop zij kunnen werken. Deze ene seinpaal- noem hem "SG", de Seinpaal Generaal- is echter al voldoende om de operatie falsify op een willekeurige gemeenschappelijke logische variabele toe te voegen aan het repertoire van alle machines. Immers:

Als wij in elke machine definieren:
"boolean procedure falsify(GB); boolean GB;
begin boolean Q; P(SG); falsify:= Q:= GB; if Q then GB:= false; V(SG) end"
en wij voorts afspreken, dat elke referentie naar een gemeenschappelijke logische variabele in elke machine ingekapseld zal worden tussen de statements "P(SG)" en "V(SG)" (zodat de assignment "GB1:= true" nu de vorm zal hebben "P(SG); GB1:= true; V(SG)", dan hebben wij hiermee verzekerd dat aan de bijvoorwaarde voldaan is, dat " gedurende de operatie falsify door één van de machines de logische variabele in kwestie ontoegankelijk is voor alle andere machines." We hebben met de ruil dus niet in eigen vlees gesneden.

Wij gaan nu laten zien, dat de operatie's P en V in hun toepassingsgebied niet beperkt zijn tot wederzijdse uitsluiting, maar dat we ze ook kunnen gebruiken, om twee machines ten opzichte van elkaar te synchroniseren. Stel, dat wij twee cyclische machines hebben, zeg X en Y, die elk in hun cyclus 'n critische sectie hebben, zeg TX resp. TY, waarbij de volgende eisen gesteld zijn:

1) de uitvoering van de critische sectie TX mag in de tijd niet samenvallen met die van TY;
2) de uitvoeringen van de secties TX en TY moeten alternerende gebeurtenissen zijn (op deze voorwaarde doelden we, toen we spraken van "onderlinge synchronisatie").

Als U zich een entourage wilt voorstellen, waarin men voor dit probleem gesteld wordt, denkt U dan maar aan het volgende. Proces X is een cijferproducerend proces (een rekenproces), proces Y is een cijferconsumerend proces (een schrijfmachine) en deze twee processen zijn gekoppeld door een "toonbank" met de capaciteit van één cijfer. De handeling TX is het neerleggen van het volgende te typen

cijfer op een lege toonbank, het stuk TY representeert hét afnemen van het cijfer van de volle toonbank. De stricte alternering vloeit voort uit de eis dat enerzijds geen enkel aan de toonbank aangeboden cijfer "weg mag raken" maar inderdaad getypt moet worden, en dat anderzijds -als er een tijdje geen aanbod is- de schrijfmachine niet zijn tijd moet gaan vullen met het nog maar eens typen van het laatste cijfer.

Wij kunnen dit bereiken met twee seinpalen, zeg SX en SY, waarbij:
SX betekent, dat nu eerst een sectie TX aan de beurt is, en
SY betekent, dat er nu eerst een sectie TY aan de beurt is.
Voor machines X en Y luiden de programma's nu:

"LX: P(SX); TX; V(SY); proces X; goto LX" en
"LY: P(SY); TY; V(SX); proces Y; goto LY" .

Wij kunnen dit direct op twee verschillende manieren uitbreiden. Indien wij de machine X vervangen door een wolkje machines Xi, die allemaal van dezelfde toonbank gebruik willen maken om hun productie te lozen, en de machine Y vervangen door een wolkje machines, die allemaal bereid zijn, om informatie van de toonbank af te nemen (ongeacht, door welke van de machines Xi het er op is gelegd, het beeld van de schrijfmachines gaat hier dus niet meer op), dan luiden de programma's voor machines Xi en Yj analoog aan de boven gegeven programma's:

"LXi: P(SX); TXi; V(SY); proces Xi; goto LXi" en
"LYj: P(SY); TYj; V(SX); proces Yj; goto LYj" .

Hadden we ook nog een groepje machines Zk gehad, en was de opgave geweest, dat de uitvoering van een TX-sectie, een TY-sectie en een TZ-sectie elkaar in die volgorde cyclisch moesten opvolgen, dan hadden de programma's geluid:

"LXi: P(SX); TXi; V(SY); proces Xi; goto LXi" en
"LYj: P(SY); TYj; V(SZ); proces Yj; goto LYj" en
"LZk: P(SZ); TZk; V(SX); proces Zk; goto LZk" .

Wij merken hierbij op, dat in de laatste twee voorbeeldjes de machines Xi gelijk zijn: zij representeren uitsluitend, dat de handelingen TX ergens door geinterspatieerd moeten worden, zij laten in het midden, hoe complex deze tussenliggende handelingen gerealiseerd zullen worden.

Een volgende opgave moge aantonen, dat onze operatie's P en V in hun huidige vorm nog niet helemaal machtig genoeg zijn. Wij beschouwen daartoe de volgende configuratie. We hebben twee machines A en B, elk met hun critische sectie TA respectieve-

lijk TB; deze secties zijn volledig onafhankelijk van elkaar, maar er is nog een derde machine C in het spel met zijn critische sectie TC en de uitvoering van TC mag in de tijd niet samen vallen met die van TA, noch met die van TB. We kunnen dit doen met twee seinpalen, zeg SA en SB. Voor machines A en B is het programma eenvoudig, nl.

"LA: P(SA); TA; V(SA); proces A; goto LA" en

"LB: P(SB); TB; V(SB); proces B; goto LB" ,

maar voor machine C is het volgende programma. hoewel veilig, onacceptabel:

"LC: P(SA); P(SB); TC; V(SB); V(SA); proces C; goto LC" .

Immers: machine C zou de statement P(SA) succesvol voltooid kunnen hebben op een moment, dat machine B in sectie TB bezig is. Zolang sectie TB nu nog niet voltooid is, kan machine C niet verder -dat is correct- maar ook machine A kan zijn sectie TA niet meer uitvoeren en dat is niet correct: indirect, via machine C zijn hier de secties TA en TB toch weer gekoppeld. Door de operatie P(SA) van machine C is de seinpaal SA overhaast op false gezet, daarmee machine A onnodig hinderend.

. Wij gaan daarom de operaties P en V uitbreiden tot operaties, die we een willekeurig aantal argumenten mee kunnen geven -daarmee de grenzen van ALGOL 60 overschrijdend, maar dat doet er in dit verband bitter weinig toe. Voor de V-operatie is de betekenis duidelijk: het is niet anders dan de simultaneous assignment, die aan alle meegegeven seinpalen gelijktijdig de waarde true toekent. De P-operatie met een aantal argumenten, bv. "P(S1, S2, S3)", is een operatie, die slechts beeindigd kan worden op een ogenblik, dat alle meegegeven seinpalen de waarde true hebben (in ons voorbeeld dus een ogenblik, waarop geldt: "S1 and S2 and S3"), beeindiging van een P-operatie houdt dan in, dat aan alle meegegeven seinpalen simultaan de waarde false toegekend wordt. Ook na deze uitbreiding wordt de beeindiging van een P-operatie weer beschouwd als een ondeelbare gebeurtenis.

Met deze uitbreiding zijn we in staat machine C zodanig te definieren, dat het gesignaleerde bezwaar niet meer optreedt:

"LC: P(SA, SB); TC; V(SA, SB); proces C; goto LC".

Tenslotte wil ik, hoewel dit beslist niet de toepassing is, waarvoor de P- en V-operaties geconcipieerd zijn, laten zien, hoe we hiermee het scalair product van twee vectoren kunnen berekenen, zonder dat we ons vast leggen op de volgorde, waarin de producten der elementen bij elkaar opgeteld worden.

Stel, dat we uit willen rekenen de som SIGMA(i,1,5,A[i] * B[i]) -in woorden: de som, voori lopend van 1 t/m 5, van (A[i] * B[i])-;

stel, dat ingevoerd zijn de 7 seinpalen: Ssom, Sklaar en de vijf seinpalen Sterm[i] met i=1,2,3,4 en 5; alle 7 met de beginwaarde false;

stel, dat inmiddels gecreeerd zijn vijf machines van de structuur (i=1,2,3,4,5):

```
"Lterm[i]: P(Ssom, Sterm[i]);
           scapro:= scapro + A[i] * B[i];
           n:= n - 1;
           if n = 0 then V(Sklaar) else V(Ssom);
           goto Lterm[i] " ,
```

alle bezig aan hun -enige- P-operatie;

Beschouw nu een zesde machine, die onder bovengenoemde voorwaarden juist zal beginnen aan de volgende statements:

```
".....; n:= 5;
       scapro:= 0;
       V(Ssom, Sterm[1], Sterm[2], Sterm[3], Sterm[4], Sterm[5]);
       "programma, dat genoemde seinpalen, scalairen en vectorelementen ongemoeid
        laat";
        P(Sklaar);...."
```

Als de laatste P-operatie van de zesde machine voltooid is, heeft n de waarde 0, scapro heeft de waarde van de gevraagde som en alle seinpalen en de vijf eerstgenoemde machines zijn terug in hun oorspronkelijke toestand.