[illegible]

On the Necessity of Correctness Proofs

As soon as someone wants to use an information processing mechanism - and in what follows we shall regard a program as "an abstract mechanism" - he wants to rely on it, that it functions properly, he wants to be convinced that the output it produces is indeed the proper function of the input. On what can this confidence be founded ?

If it is a very simple piece of machine - e.g. when it accepts two one-decimal numbers and is requested to produce the two-decimal product - we could try all 100 different multiplications the machine claims to be able to perform and check all the answers, a so-called exhaustive test.

In the case that we are forced to consider the mechanism as a black box, an exhaustive test is the best we can do. (And even if we are able to perform an exhaustive test, this is not fully conclusive : we must make the assumption that at each application the output is only a function of the current input and not of past history. The absence of interior memory elements that could record such past history can never be established by such experiments.)

But even single machine instructions - such as the fixed-point multiplication of two word-size integers - are already defined on such a gigantic domain, that an exhaustive test is absolutely out of the question : even for very fast machines it is not unusual that the total time taken by all possible activities evoked under control of the multiply instruction will be well over a million years. Are exhaustive tests out of the question for individual machine instructions, then this holds a ~~function~~ fortiori for complete programs, where the domain is usually orders of magnitude more time-consuming.

./

As long as we continue to regard the mechanism as a black box tests are the only things to which we can subject it. We could raise our confidence by sampling on a statistical basis, but the virtual number of possible test runs is so negligibly small compared with the number of possible computations that not much confidence can be gained that way : whole classes of in some sense critical cases can and will be missed. The moral of the story is, that we cannot continue to regard the mechanism as a black box : we must open it and in some way or another must take its construction, its interior structure, into account.

A usual way is to verify by experiment that a modest set of cases actually works correctly, from which we then conclude on account of its structure, that all cases must work correctly. (E.g. quality control of a camera, which does not force the manufactures to make "all possible pictures" with a camera before it can be delivered !).

In the case of programs, however, a few remarks are in order :

1. In general we can test far less than one out of a billion cases : for nearly all cases we have to rely on our reasoning. In the case of programs, discrete and abstract mechanisms, not subject to wear and tear, it is not clear at all how we can benefit by not requiring that the number of test runs still needed, is reduced to zero. On the contrary, it seems more straightforward not to rely on testcases at all and to prove the program's correctness a priori.

2. One of the reasons that in many programming circles correctness proofs are not fashionable is that for an arbitrary program the amount of formal labor to supply such a proof can be quite enormous. The necessary amount of formal labor, however, is critically dependent on the structure of the program, and it is here that correctness concerns have a strong feedback on the programs to be produced : a major function of the structuring of the program is to keep a correctness proof feasible.

./

On the Mathematical Structure of Correctness Proofs

Correctness proofs can only be given, provided that the semantics of our programming language are given by a formal definition, and for a long time the absence of a good technique for giving such a formal definition has prevented correctness proofs : there was no foundation to build upon.

Anyone familiar with the Report on ALGOL 60 will agree that, compared to earlier efforts at language definition, the introduction of BNF (Backus Naur Form) was a tremendous step forward as far as the language syntax was concerned. the semantics, however, were given in - be it : carefully phrased - English. And for a couple of years that was that.

Since then, I have seen efforts falling into three main classes. The first class we can call "the mechanistic definition". Here the semantics of a programming language are given in terms of an "interpreter", written for an "abstract machine", the idea being that the abstract machine - although utterly unrealistic - could be so simple, that no missunderstanding would be necessary with respect to its "order code", and once you have grasped this, you only need to "follow" the interpreter if you want to know the output of a given computation. (The Vienna Definition Language for PL1 and also ALGOL 68 fall into this class). Such mechanistic definitions, however, have a few serious drawbacks. There is the fundamental shortcoming that we are still faced with the problem of formally defining the semantics of the abstract machine, and in that sense, the problem has not been solved, it has only been pushed back. A more serious shortcoming is that, initially, the interpreter can only tell you what the output will be of a specific computation : you just play the game, whereas the kind of assertions we would like to make about programs are assertions about the class of all computations that can be evoked under control of that program. Finally as the semantics is defined in terms of an (idealized) implementation, the problem of the correctness of a realistic implementation amounts to the equivalence of programs, a hairy problem if there ever was one.

./

It is not surprising that the effort to build upon a mechanistic definition presented grave difficulties and my impression is that with the exception of a few groups that have committed themselves, the effort has been abandoned.

The next effort tries indead to capture all possible computations that can take place under control of a program, by stating axiomatically how the "output" is functionally dependent on the "input". This function is defined as a (specific) solution of a functional equation which can be derived mechanically from the program text. Such is - in very rough terms - the approach that has been started by Dana Scott. In this mathematical foundation lattice theory and functional analysis play a predominant role : whether it will turn out to be a useful tool remains to be seen. Its outstanding characteristic is that the notion of "output" has been extended to non-terminating algorithms.

The third effort has been originated by C.A.R. Hoare, who has given an axiomatic definition in terms of rules for deriving for a given piece of program for any post condition to be satisfied after execution of the program the weakest pr~~a~~econdition for the initial state. The idea is that if we can do this for any postcondition, that then we know all about the semantics of the program.

Let the weakest praecondition for a given program S and some postcondition P be denoted by fS(P). If fS(P) = T (i.e. identically true), then the program is correct, if fS(P) = F (i.e. identically false), then it is wrong, in all "in between cases" it establishes the relation P as a partial function. Let me show for a simple language, how the method works.

./

Axiom of Assignment :

If S is of the form "X := E" then fS(P) : P_X^E , where P_X^E is obtained by replacing all occurences of X in P by the expression (E).
(e.g. with S of the form "X := X + 1"

$fS(a < 7) = a < 7$
$fS(X < 10) = X < 9$).

The Axiom of Assignment gives us the condition-transformer fS for all programs consisting of a single assignment statement. To derive the condition transformer for more complicated programs we postulate, how the total condition transformer is formed in terms of the condition transformers of the components and the sequencing connective.
For the semicolon, we have the

Axiom of Concatenation

If S is of the form "S1 ; S2", where the semantics of the components S1 and S2 are given by the condition transformers fS1 and fS2 respectively, then fS(P) = fS1(fS2(P)).
From this it follows that the operation of statement concatenation is associative, i.e

(S1 ; S2) ; S3 = S1 ; (S2 ; S3)

The Axiom of Alternatives :

If S is of the form "if B then S1 else S2 fi"
then fS(P) = (fS1(P) and B) or (fS2(P) and non B)

The Axiom of Repetition :

If S is of the form "while B do S1 od"
then $fS(P) = (\exists i : i \geq 0 : fS1^i(P \text{ and non } B) \text{ and } (\forall j : 0 \leq j < i : fS1^j(B)))$
where $fS1^0(P) = P$ and $fS1^{i+1}(P) = fS1(fS1^i(P))$.

./

These condition transformers can be showh to satisfy four absolutely basic properties :

Property 0 : $P = Q$ implies $fS(P) = fS(Q)$

Property 1 : $fS(F) = F$

Property 2 : $fS(P$ and $Q) = fS(P)$ and $fS(Q)$

Property 3 : $fS(P$ or $Q) = fS(P)$ or $fS(Q)$

(Property 1 is called : the law of the Excluded Miracle).
And it is on account of these four properties that we are allowed to interpret any $fS(P)$ we form as the weakest praecondition for a deterministic automaton with P as postcondition.

To give a simple example, let S be

if $X < 40$ then $X := X + 20$ fi

(Here S2 is the empty statement, i.e. $fS2(P) = P$) then $fS(P) = P_X^{X+20}$ and $X < 40)$ or $(P$ and $X \geq 40)$ and for instance
$fS(X \geq 50) = X + 20 \geq 50$ and $X < 40)$ or $(X \geq 50$ and $X \geq 40) = (30 \leq x < 40)$ or $(x \geq 50)$.

The real trouble arises with the application of the Axiom of Repetition : unless we can find a closed expression for $fS_1^i(P)$, i.e. for the i.th iterative of S1, direct application of the axiom of repetition presents us with a hairy problem, as untractable as the problem of finding a closed expression for the limit of an infinite series may be, when a recurrence relation between successive terms is given. Our moral is that there is no point in writing down "while B do S od" for totally unrestricted B and S : in general we have written down something entirely unmanageable? But from the axioms we may derive the following theorem :

./

Let S be "while B do S1 od" ; let for some P and R hold
fS1(P) =(P and B) or R (or, as we may also write, let (P and B) → fS1(P)), then there exists a relation Q, such that fS(P and non B) = (P and fS(T)) or Q (or (P and fS(T)) → fS(P and non B).
In words : fS(T) is the weakest praecondition such that the repetition will terminate ; then P and fS(T) is a sufficient initial condition such that the loop will terminate such that P and non B is guaranteed to hold upon termination.

The power of this theorem is that the initial condition is written as a conjunction of two conditions, one concerned with termination only (which is independent of the target condition P) and another, which is concerned with the invariance of P.
The fact that a single execution of S1 will not destroy the validity of P and that upon termination of the loop P will still hold, regardless the number of times the repeatable statement has been executed, is the basis for the so-called "strategic abstraction", which enables us to map different loops upon each other, differing in the number of times that they will be executed, but leaving the same relation invariant. If we compare

```
   q := a ; c := 1 - b ;
   while abs(c) > eps do a := a × (1+c); c := $c^2$ od
```

with

```
   q := a ; c := 1 - b ;
   while abs(c) > eps do a := a × (1 + c + $c^2$) ; c := $c^3$ od
```

both programs maintain the invariance of

$$\frac{a}{b} = \frac{q}{1 - c}$$

and for abs(c) < 1 they will both squ~~are~~eeze c towards zero, and both can be used to approximate a/b without actually dividing : both loops are of the form "while c too large do squ~~ar~~eeze c under invariance of q/(1-c) od".

./

When we are faced with the task of writing a program to find the convex hull for a number of points in the plane, many different programs are possible, but they are all refinements of the following general pattern :

```
"construct the convex hull for a few-say two or three-points,";
while there are still points outside the current hull do select a
points outside the current hull ;
adjust the hull to enclose the selected points as well
od
```

and the only difference I could detect among the different algorithms, was the "shrewdness" with which they selected the "initial point" and the "next point outside". At this level of details, the invariance enables us to consider a program in which the actual number of repetition is unknown on account of the non-deterministic primitive (the "gambling" primitive) "select a point outside the current hull". It is only by virtue of the invariance theorem that we gain by considering non-deterministic machinery ; conversely when faced with non-deterministic machinery - as in the case of cooperation between sequential process with undefined speed ratios - invariance is the thing to look for.

When two sequential processes with undefined speed ratios have to be synchronized, we can regard this as controlling the process path in a two-dimensional progress space. The classical example is the preparation of two dishes, soup and stew, both using a ring on the stove and a mixer. The individual usage pattern of these resources are for the soup :

```
claim a ring ;
claim a mixer ;
release the ring ;
release the mixer ;
```

and for the stew

```
claim a mixer ;
claim a ring ;
release the mixer ;
release the ring.
```

./

When the two dishes have to be prepared in a one-ring kitchen, this restriction declares a rectangle in progress space - viz. the area interpretable as "both using the ring" as disallowed area. A similar rectangle can be shown when the two dishes have to be prepared in a one-mixer kitchen. If, however, both restrictions are imposed simultaneously, we see that the total disallowed area is larger than just the superposition of those two rectangles : when we allow the soup to get the ring, but not yet the mixer and the stew to get the mixer but not yet the ring, then we see that although the original restrictions have not been violated yet, we are hopelessly stuck : they are stuck in a "deadly embrace", we have created a deadlock situation. The well-known moral of this part of the story, is that in order to exorcize the danger of deadlock, we must be willing (and able !) to extend the primarily forbidden area in progress space with the "traps" in order to make it well-shaped. In general this is a very tough job, as difficult as proving that an arbitrary loop does (or does not) terminate.

When our forbidden area is well-shaped, the ensuing control problem can be solved, and in a later example we shall show how this can be done. Before we tackle this example, however, the method will be extended to cover a wider class of restrictions.

Besides restrictions that can be phrased in terms of forbidden areas in the progress space, we may have to cater for restrictions formulated in terms of the shape of the path. Such restrictions can be translated in restrictions of the previous type by a perfectly straightforward manner, that goes as follows. To our state space in which (place and) shape restrictions on the path are given, we add a further dimension in which we record, at any moment in time, enough data about the history so that in terms of the position in this extended space, the total restrictions can be formulated as "a disallowed volume". In this extended space the primarily disallowed volume is extended with the traps in that multidimensional space and in that multidimensional space the control problem is solved in the usual manner.

./

To give an example of how this works : consider an ant that has to crawl on a flat surface from point A to point B, while the total path length is not allowed to exceed a given upper bound L. We then associate with the ant a butterfly with a twofold restriction in its movements :

1. it must remain perpendicularly above the ant - the ant's position is the vertical projection of the ant ;
2. it is bound to ascend always under 45 degrees.

If both ant and butterfly start their journey at point A, these conditions guarantee that the distance crawled by the ant is equal to the height of the butterfly and the restriction on the ant's path can be translated into the requirement that the butterfly is not allowed to rise above a horizontal ceiling at height L. As the ant has to arrive at point B, the butterfly has to arrive at the perpendicular erected at point B.
In this case the three dimensional trap of the butterfly is the space outside the cone with its top at the point where the perpendicular in B cuts the ceiling at height L, which cuts the ant's surface in a circle with centre B and radius L. If A lies inside this cone, the ant's problem can be solved : as it crawls, the butterfly rises, until it hits the cone : from that moment onwards, the butterfly has only one possible path inside the cone and satisfying the 45 degree requirement, i.e. straight to the top.

In a system like this, two relations must be kept invariant

1. the height of the butterfly must remain equal to the distance travelled by the ant,
2. the butterfly must remain inside the cone.

In the case of a multi-dimensional control problem corresponding to some synchronisation task the "state of affairs" is described by a number of variables, that can be inspected in order to establish whether a desired "move" is permitted - i.e. will not violate the relations to be kept invariant - and can be modified in order to represent that the move has taken place.

../

Let Si be such a move, let Bi be the condition under which it can take place. The inspection whether a condition Bi holds and, if so, the subsequent execution of Si, are occurences that should exclude each other in time, i.e. no Bj - Sj succession should be allowed to tamper simultaneously with those common state variables. This mutual exclusion is the basis for our conclusion that, when individual steps maintain the invariant relations, these relations will indeed continue to hold. Without the mutual exclusion we would be forced to consider the net effect of all coincidences of pairs, triples, etc as well ! In a later stage we shall return to these matters.

./

On the Feasibility of Correctness Proofs

All this is all very well, but it is no good unless we manage to arrange our thoughts in such a manner, that the amount of reasoning necessary to convince ourselves of the correctness of a program does not explode with program length ! For if that is the case, we shall never be able to apply these methods to anything beyond toy problems.

A first instance we have seen at the end of the previous section. If in a parallel programming environment, N operators can fool in a common state space, mutual exclusion of these operators makes it sufficient to study them individually : without the mutual exclusion we should have to study the total effect of all combinations, and something like 2^N cases emerge !

But even in sequential programming, something of that sort emerges. Consider a piece of program of the form :

```
"if B1 then S11 else S12 fi ;
 if B2 then S21 else S22 fi ;
             :
 if Bn then Sn1 else Sn2 fi "
```

If we take for granted that - we have to introduce some sort of measure ! - that it takes two steps of reasoning to equate

```
"if Bi then Si1 else Si2"
```

to an abstract statement "Si", then in 2, steps we have reduced our program to an abstract program of the form

```
"S1 ; S2 ; .... Sn"
```

of which I will assume that it takes us another n steps to equate it to a total program S. In this unit, it takes us 3n steps of reasoning.
If we had insisted in understanding the total computation in terms of a succession of the individual Si1's Si.2's, each such succession would need n steps for its understanding, but there are 2^n possible ways of sequencing ! The introduction of the abstract statements Si prevents this exponential growth !.

./

The above observation is older than Hoare's axiomatic method for the definition of the semantics : I would like to point out that the introduction of the abstract statements Si is exactly what one is lead to, as soon as one tries to derive, according to Hoare, the "fS" for the total program ! In that respect Hoare's axiomatic basis seems a sound one.

If the method is to work at all, the fSi, as derived from Bi, fSi1 and fSi2, should not be too unwieldy: the net effect of the conditional compound should lend itself to a clear and compact formulation.
As soon as we are not able to do so, this is a warning not to be ignored : probably we are on the verge of messing things up !.

The "statement grouping" as suggested by Hoare's axioms is called "operational abstraction". It has - as all our abstraction patterns - a dual purpose : firstly, it enables us to reduce the amount of labor involved in the understanding of a specific program, secondly, because this piece of reasoning applies to an abstract program, as a rule admitting various   we have that alternative programs for the same task - i.e. different refinements - may share part of the correctness proof.

In connection with the convex hull we have mentioned "strategic abstraction", finally we would like to show an application of representational abstraction.

Suppose that it is required, for global integer A($\geq$1), B($\geq$0) and Z, to program the assignment

$$Z := Z * A^B$$

with the aid of an inner block, not using exponentiation. A very useful overall pattern of inner blocks is the following.

./

At block entry a local variable is introduced and initialized in such a way that some relation between inner and outer world holds. From then onwards, inner and outer world are "massaged" under invariance of that relation, until the local variable has a non-interesting value, and block exit follows.

Four our purpose, we introduce at level 1 an unanalyzed local variable "h" and write level 1 :

```
   local h initialized such that P1 holds ;
   while h unequal to one do squeeze h under invariance of P1 od
```

where, if Z' denotes the initial value of Z, relation P1 is

$$Z * h = Z' * A^B$$

and the squeezing operation, when applied to a value of $h \neq 1$, is guaranteed to make $h = 1$ in a finite number of applications. The invariance of P1 and the final value of h guarantee at the end $Z = Z' * A^B$, as desired.

In our next level we wish to refine these operations in sub-operations, referring to either global or local world, such that P1 is guaranteed to hold initially $Z = Z'$, and the initialization becomes

level 2 : local h initialized such that $h = A^B$;

while at that same level the operation "squeeze etc " becomes

```
level 2 : set integer f to a factor by which h is divided ;
          multiply Z by f.
```

It is the function of this level to separate operations on the local h and the global Z, thereby maintaining the relation P1. We observe that internally (at the semi colon) the relation P1 is temporarily destroyed.

./

Now the time has come to choose a proper representation for h. As the absence of the exponentiation was the primary reason for this inner block, a single variable of type integer won't do for h. We therefore introduce representation convention :

$$h = X^y$$

and the initialization becomes

```
level 3 : integer X = A, y = B ;
```

the test "h unequal to one" becomes

```
level 3 : X ≠ 1 and y ≠ 0
```

and the setting of f

```
level 3 : while even (y) do y := y/2 ; X := X * X od ;
          y := y - 1 ; f := x
```

(letting f and x coincide, we could merge the three levels into the ALGOL block :

```
   begin integer X,y ; X := A ; y := B ;
         while X ≠ 1 and y ≠ 0 do
            while even (y) do y := y/2 ; X := X * X od ;
            y := y - 1 ; Z := Z * x
         od
   end
```

The test " X ≠ 1" could have been omitted).

We summarize :

in level 1 the invariance of P1 is exploited without detailed knowledge about the representation of either the outer, nor the inner world ;

in level 2 the invariance of P1 is catered for, under the assumption that operations on outer and inner world can communicate via the standard type "integer" ; level 2 assumes the availability of a proper representation of h ;

in level 3 a representation of h is choosen and the assumptions made in level 2 about h are catered for.Relation P1 is here of no concern.

./

The assumption about the outer world are
1. its ability to deliver the values A and B upon request
2. its ability to multiply Z by the value f

The purpose of this exercise was many fold. We know that, by grouping statements, we can regard computational processes with different grains of time. Also we know, that we can group words in store : we can regard the state in different grains of space. One of the purposes was to make the different grains of space and time "interlocking" : it is for that reason that we have introduced levels 1 and 2, where "h" is still regarded as a non-analyzed abstract variable of some suitable type.

This seems essential : a program of a high level of abstraction should be understood in terms of sufficiently abstract variables, and not in terms of a specific elaborate representation for the different possible values of such a variable.

It also shows how - at the expense of a "communication level" such as level 2 - we can separate the operations on the local h (represented in some appropriate fashion) and the global Z (also represented in some appropriate fashion), where we can do this for the price of a commonly understood more primitive type (here the integer f).

So many programs slowly grow into a mess of conflicting conventions that can no longer be disentangled. For that reason, it seems a worthy goal to encapsulate in the system write-up the consequences of each particular convention. It is clear that we can never make a program out of modules each with its own conventions, for they can only communicate via a common convention. It is suggested that many of such conventions take the form of an invariant relation, upon which one module may rely, whereas it is another's module's obligation to guarantee it.

./

On the Impact of Correctness Concerns on "the Process of Program Composition"

The purpose of this section (as a sequel to the second section) is to show a more elaborate example of program composition, as it can be controlled by the Hoare formalism of the weakest preconditions.
We have a set of cyclic processes, called Readers and Writers, respectively, with a critical activity, called "read" and "write" respectively, and they should be synchronized in such a way that

a) a reader doing "read" does not exclude other readers doing "read", but all writers from doing "write" ;

b) a writer doing "write" excludes all readers from doing "read" and all other writers from doing "write".

I assume the programs to have the following structure :

```
cycle remainder ;                 cycle remainder ;
      READENTRY ;                       WRITEENTRY ;
      read ;                            write ;
      READEXIT                          WRITEEXIT
elcyc                             elcyc
```

Where a neutral mutual exclusion mechanism for the four operations denoted in capital letters - in order to prevent uncontrolled interferences - will be assumed. In order to be able to formulate our requirements, we introduce two counters ar and aw (active readers and active writers), with initial values = 0. If we now state

```
S1 : READENTRY : ar := ar + 1
S2 : READEXIT  : ar := ar - 1
S3 : WRITEENTRY : aw := aw + 1
S4 : WRITEEXIT  : aw := aw - 1
```

then our basic relation becomes :

P(ar, aw) : $(ar \geq 0$ and $aw = 0)$ or $(ar = 0$ and $aw = 1)$

./

given [illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible] P [illegible]

From the topology of our programs it follows that upon READEXIT $ar > 0$ must hold, therefore, its "ar := ar - 1" can never cause violation of P, and also that upon WRITEEXIT, $aw > 0$ must hold, therefore on account of P, $aw = 1$ must hold and also WRITEEXIT can never cause violation of P.

The entries, however, can cause violation and here we must construct the condition B1 and B2 upon which they may take place. According to the axiom of assignment $C1 = P(ar + 1, aw)$ and $C2 = P(ar, aw + 1)$, i.e.

C1 : $(ar + 1 \geq 0$ and $aw = 0)$ or $(ar + 1 = 0$ and $aw = 1)$
C2 : $(ar \geq 0$ and $aw + 1 = 0)$ or $(ar = 0$ and $aw + 1 = 1)$

Because we know that P will hold, whenever the investigation is made, we can simplify these expressions replacing C1 by the simplest expression B1, such that P and $B1 \Rightarrow C1$, and similarly P and $B2 \Rightarrow C2$.

When faced with the task to find, for given P and C a simple B such that P and $B \Rightarrow C$, we use two theorems :

1. if "B = Q or R" is a solution and P and R is false, then Q is a solution
2. if "B = Q and R" is a solution and $P \Rightarrow R$ then Q is a solution.

For B1 our first solution is C1 ; on account of theorem 1, it can be reduced to

$ar + 1 \geq 0$ and $aw = 0$

and on account of theorem 2 it reduces to B1 : $aw = 0$

For B2 our first solution C2 reduces on account of theorem 1 (theorem 2 is then not applicable) B2 : $ar = 0$ and $aw = 0$

./

Assuming the proper mutual exclusion we can write

```
   READENTRY : when aw = 0 do ar := ar + 1 od
   WRITEENTRY : when ar = 0 and aw = 0 do aw := aw + 1 od
```

meaning that no reader will be waiting when aw = 0 and no writer when ar = 0 and aw = 0. If we want to express this in terms of variables manipulated by the program, we must introduce additional variables, say br and bw (blocked readers and writers), counting the number of waiters. If we initialize them both to 0, we can write

```
   READENTRY : br := br + 1 ; (ar := ar + 1; br := br - 1)
   WRITEENTRY : wr := wr + 1 ; (aw := aw + 1 ; bw := bw - 1)
```

where the parts within the parentheses are the conditional actions, to be executed such that P' = P and br ≥ 0 and bw ≥ 0 be kept invariant.

A second remark is that WRITEREXIT (aw := aw - 1) that will cause aw = 0 may make both READENTRY and WRITEENTRY possible. We now superimpose the requirement, that when writers will get priority, no reader may be admitted, when there is a writer waiting.

If we want to follow the formal game, we introduce an explicit counter V (also initialized to 0), counting the violations,and rewrite

```
   READENTRY : br := br + 1
               (ar := ar + 1 ; br := br - 1 ;
                if bw > 0 then V := V + 1)
```

and now imposing the total invariant relation P" = P' and V = 0

With the new forms of the entries and P", the weakest praeconditions for the conditional parts become for the READENTRY :

```
   C1 : ((ar + 1 ≥ 0 and aw = 0) or (ar + 1 = 0 and aw = 1)) and
             br - 1 ≥ 0 and ((bw ≤ 0 and V = 0) or (bw > 0 and V + 1 = 0))
   C2 : ((ar ≥ 0 and aw + 1 = 0) or (ar = 0 and aw + 1 = 1) and bw - 1 ≥ 0.
```

which, because P" is kept invariant can be reduced to

```
   B1 : aw = 0 and br > 0 and bw = 0
   B2 : ar = 0 and aw = 0 and bw > 0
```

./

Now we are in good shape because (on account of the last term) they exclude each other, and therefore we have no choice any more.

Because it will be the function of the ENTRY's and EXIT's to ensure

1. that the conditional do not take place when they would violate P"
2. but that they will take place when they don't violate P", i.e. when B1 or B2 holds.

We can conclude that outside EXIT's and ENTRY's we can make the much stronger assertion

P" <u>and</u> <u>non</u> B1 <u>and</u> <u>non</u> B2.

With semaphores (special purpose non-negative integers), the P and V operation (where the V operation increases a semaphore by 1 and the P operation - representing a potential delay -) we can now rewrite our programs with three semaphores : R and W (initially 0) and mutex, initially 1.

The programs then take the form :

```
cycle remainder ;                cycle remainder ;
      READENTRY ;                      WRITEENTRY ;
      P(R) ;                           P(W) ;
      read ;                           write ;
      READEXIT                         WRITEEXIT
elcyc                            elcyc
```

and in the text of the four critical sections we can exploit that upon entry

P" <u>and</u> <u>non</u> B1 <u>and</u> <u>non</u> B2

holds.

./

Without exploitation of that knowledge, we could write

```
READENTRY :
    P(mutex) ; br := br + 1 ; TEST ; V(mutex) ;
READEXIT :
    P(mutex) ; ar := ar - 1 ; TEST ; V(mutex) ;
WRITEENTRY :
    P(mutex) ; bw := bw + 1 ; TEST ; V(mutex) ;
WRITEEXIT :
P(mutex) ; aw := aw - 1 ; TEST ; V(mutex);
```

where TEST :

```
While B1 or B2 do
    if B1 then ar := ar + 1 ; br := br - 1 ; V(R)
          else aw := aw + 1 ; bw := bw - 1 ; V(W)
    fi
          od
```

making obvious that P" is never violated and non B1 and non B2 will hold upon exit of the critical section.

Exploitation of the initial invariant reduces the number of necessary tests

```
READENTRY :
    P(mutex) ;
      br := br + 1 ;
      if aw = 0 and bw = 0 then
      ar := ar + 1 ; br := br - 1 ; V(P) fi
    V(mutex)
READEXIT :
    P(mutex) ;
      ar := ar - 1 ;
      if ar = 0 and bw > 0 then
      aw := aw + 1 ; bw := bw - 1 ; V(W) fi
    V(mutex)
```

./

```
WRITEENTRY :
    P(mutex) ;
      bw := bw + 1 ;
      if ar = 0 and aw = 0 then
      aw := aw + 1 ; bw := bw - 1 ; V(W) fi
    V(mutex)
WRITEEXIT :
    P(mutex) ;
      aw := aw - 1 ;
      if bw > 0 then aw := aw + 1 ; bw := bw - 1 ; V(W) fi
                 else
      while br > 0 do ar := ar + 1 ; br := br - 1 ; V(R) od
    V(mutex)
```