Copyright Notice

A personal summary of the Gries-Owicki Theory.

This is a very personal summary of the theory developed by Susan Speer Owicki under supervision of David Gries. I had a flu, and on its first day I just slept and shivered; later I passed the time in bed with trying to reconstruct what I had learned from reading in Susan Owicki's doctoral thesis. If the following fails to do justice to their work --someone has borrowed my copy of her thesis!-- I am the only one to blame.

There has been a time that it was the function of our programs to instruct our machines, but times have changed: now it is more fruitful to consider it the purpose of our machines to execute our programs. The same shift of attention can be recognized in the more theoretical work that is concerned with the semantics of programming languages. There has been a time, that this was a very descriptive activity, trying to capture what happened in our machines during program execution. The result has been a series of operational language definitions. in which the semantics of programming languages was given via an interpreter that under control of the program text changed the machine state over and over again. By means of "abstract programs" and equally "abstract states" people have tried to mold this approach into a viable tool, but it kept all the essential disadvantages of operational language definitions. Faced with a specific program they tell you no more than how to do a hand-simulation. Since Floyd, and later but more noticeably Hoare, we have been shown another approach, which seems more promising.

Here a program text is regarded as a mathematical object all by itself, which is postulated to establish a relation between two machine states. If we were very pure, we should call them, say, the "left-hand state" and the "right-hand state". The relation between the two states is implicitly given by a set of axioms and rules of inference that together delineate what, given a text, one can prove about that relation. Taken all by itself, this would be a very formal and rather sterile game, but it so happens that the axioms have been chosen very carefully, so carefully in fact, that when we identify the "left-hand state" with the initial state and the "right-hand state" with the final state of a computer (as can be recorded in its store) a started sequential computer can establish an instance of that relation (and even can do so without implicit backtracking).

In the preceding paragraph I have tried to capture the essence of this so-called "axiomatic method" as clearly as possible, because it has generated after its introduction much misunderstanding and discussion (which has generated more heat than light). Even as much as five years later the axiomatic method has been blamed for not demonstrating that it captured correctly the computational model, that was supposed to underly it,"the computational model on which it was based". The axiomatic method is not "based" upon a computational model, the most we can say is that it has been inspired by a computational model. Once the axioms are chosen, it is the obligation of the implementation to provide a sufficiently truthful model. With purely sequential programs, this approach has been very successful, the Gries-Owicki Theory presents the first significant step towards applying similar techniques to concurrent processing as well.

Taken literally, the previous sentence makes no sense. From a very puristic point of view, neither Floyd, nor Hoare (nor I in the early seventies) talked about "sequential programming" or "sequential programming languages". We talked about texts, and about proving things about them. The aspect of "being sequential" had absolutely no meaning on that level of discourse, it became only meaningful when we tried to visualize a computer establishing an instance of the relation, when we tried to visualize "a computation". And the axioms we considered were such that the only safe and realistic implementation of such a computing engine that we could envisage, was one in which the actions took place one after the other. Apart from that "implementation detail" the whole notion of sequentiality was not applicable in our level of discourse in which we had abstracted quite rigorously from the class of computational histories.

From the same puristic point of view, the Gries-Owicki Theory does not deal at all with concurrent processing. It is again a formal system relating a pair of machine states to each other by means of a text. Only the proof rules --the axioms and the rules of inference-- differ. It so happens that, when we would like to design a computing engine able to establish an instance of this relation, we suddenly see a straightforward way in which a number of processors could be engaged concurrently on that task. So we are not designing a "language for concurrent programming" or any similar misnomer, from our mathematical point of view it is a programming language as any other, with consequences and possibilities for the implementation that we should ignore at the current level of discourse.

A simple "sequential" program can be represented as

S = "S0; S1; ... ; Sn" .

When we wish to describe in more detail the kind of relations between initial an final state, say that we wish to establish a set of initial states corresponding to a final state satisfying the relation R , we can interlace our sequence of statements Si with a sequence of relations Pi :

"{P0} S0; {P1} S1; ... ; {Pn} Sn {R} .

The axiomatic definition associates with each statement Si --assignment statements to start with-- a so-called predicate transformer wp . If now we have

for $0 \leq i < n$ Pi = wp(Si, Pi+1)

Pn = wp(Sn, R)

then, for the whole program S we have P0 = wp(S, R) and we interpret P0 as the weakest pre-condition for the initial state such that starting program S as a whole is certain to end up in a final state satisfying R .

This is, because from given units Si --say: assignment statements-- the semicolon describes how a new unit can be formed. In formula the semantics of the semicolon is given by

wp("S1; S2", P) = wp(S1, wp(S2, P))

from which, for instance, follows that the semicolon is associative. If we wanted, for instance, to combine in program S the first two initial statements to a single unit --indicated by square brackets-- we could indicate this as follows:

"{P0} [S0; S1]; {P2} S2; ... ; {Pn} Sn {R}"

By combining S0 and S1 in the above way into a single unit, the relation P1 remains anonymous; implementation-wise it says that we prefer not to pay explicit attention to the "intermediate state" that will prevail after the execution of S0 , but before the execution of S1 . In the purely "sequential systems" we are familiar with, our freedom in combining units into larger ones, thereby eliminating the "internal predicates" is unrestricted: we are all the time free to choose to consider a composite object either as an unanalyzed whole or as something composed out of parts. In the Gries-Owicki Theory this freedom is restricted (thereby giving the implementation greater freedom, such as the introduction of concurrency).

We have shown on the previous page how the concatenation via the semi-colon gives rise to internal predicates. So do the other sequencing techniques or "control structures" in the case of "sequential programming" , e.g.

$$\{P12\}\ \underline{\text{if}}\ B1 \rightarrow \{P1\}\ S1$$
$$[\!]\ B2 \rightarrow \{P2\}\ S2$$
$$\underline{\text{fi}}\ \{R12\}$$

Here the rules are that P12 should be the weakest predicate satisfying

$$P12 \Rightarrow (B1\ \underline{\text{or}}\ B2) \quad \text{(in order to avoid abortion)}$$
$$(P12\ \underline{\text{and}}\ B1) \Rightarrow P1$$
$$(P12\ \underline{\text{and}}\ B2) \Rightarrow P2$$

where P1 and P2 are given by $P1 = wp(S1, R12)$ and $P2 = wp(S2, R12)$. Again we are free to "eliminate" predicates such as P1 or P2 , for instance by replacing the second equation by

$$(P12\ \underline{\text{and}}\ B1) \Rightarrow wp(S1, R12) \quad .$$

In our program we could indicate that elimination of P1 for instance by

$$\{P12\}\ \underline{\text{if}}\ [B1 \rightarrow S1]$$

somehow suggesting that the whole first guarded command is to be regarded as a single unit. In "sequential programming" such freedom of combination, of elimination of predicates, is unrestricted.

The notation of the square brackets is unattractive if we want to indicate the elimination of the predicate following a repetitive construct. The second problem that the repetitive construct introduces is the problem of termination. Provided

$$(P12\ \underline{\text{and}}\ B1) \Rightarrow wp(S1, P12)$$
$$(P12\ \underline{\text{and}}\ B2) \Rightarrow wp(S2, P12)$$

we can read and justify

$$\{P12\}\ \underline{\text{do}}\ B1 \rightarrow S1$$
$$[\!]\ B2 \rightarrow S2$$
$$\underline{\text{od}}\ \{P12\ \underline{\text{and}}\ \underline{\text{non}}\ (B1\ \underline{\text{or}}\ B2)\}$$

as stating that the initial validity of P12 is sufficient to ensure the final validity of $(P12\ \underline{\text{and}}\ \underline{\text{non}}(B1\ \underline{\text{or}}\ B2))$, provided that the repetitive construct terminates on this level. If the repetitive construct is followed by a next statement, we can again eliminate its post-condition by straightforward proving that it implies the pre-condition for that following next statement.

Certain predicates are never eliminated. We never eliminate the predicate describing the total pre-condition, nor the predicate describing the total post-condition. (In a sense they can never be regarded as the internal predicate of a composition.) Furthermore we shall never eliminate what could be described as "the post-condition of a guarded command set". If the guarded command set is the body of an alternative construct, this refers to the post-condition of the alternative construct; if the guarded command set is the body of a repetitive construct, this refers to the invariant relation. The reason for this restriction is the following: each assignment statement and each set of guards has now a unique preceding predicate, where with "preceding predicate" we mean the last preceding, non-eliminated predicate. For instance

```
{P0} S1; S2;
{P1} S3; if B4 → {P2} S4
         [] B5 → S5; S6
        fi;
{P3} S7;
{P4} do B8 → S8; {P5} S9 {P4}
     [] B10 → S10 {P4}
    od; S11 {R}
```

Then we have:

P0 is the preceding predicate of S1, and S2;

P1 is the preceding predicate of S3, B4, B5, S5, and S6;

P2 is the preceding predicate of S4

P3 is the preceding predicate of S7

P4 is the preceding predicate of B8, S8, B10, S10, and S11

P5 is the preceding predicate of S5.

Besides non-abortion in the alternative construct and termination of the repetitive construct, we have to prove

P0 => wp(S1, wp(S2, P1))

P1 => wp(S3, (B4 => P2) and (B5 => wp(S5, wp(S6, P3))))

P2 => wp(S4, P3)

P3 => wp(S7, P4)

P4 => (B8 => wp(S8, P5)) and (B10 => wp(S10, P4)) and (non (B8 or B10) => wp(S11,R))

P5 => wp(S9, P4) .

Here are six relations. They are implications with an assertion at the left-hand side, and at the right-hand side, besides other assertions, only guards and statements of which the left-hand side is "the preceding predicate".

Suppose for a moment that, via other means we have established that P0 is strong enough to guarantee proper termination as well. Starting the obvious sequential implementation in an initial state satisfying P0 , a computation would ensue during which at the corresponding stages the machine would be in a state satisfying one of the Pi's, and finally the machine would end in a state satisfying R. What would we have to prove in addition if we would like to ensure, that at all those stages another predicate, Q say, would be true as well? This, of course, under the assumption that we would start the machine in an initial state also satisfying Q .

Well, in principle, we should replace in our six relations all the predicates Pi and R at all their occurrences by Pi and Q and R and Q respectively! The first line would then become

P0 and Q => wp(S1, wp(S2, P1 and Q))

Its right-hand side reduces as follows:

wp(S1, wp(S2, P1 and Q)) = wp(S1, wp(S2, P1) and wp(S2, Q))

= wp(S1, wp(S2, P1)) and wp(S1, wp(S2, Q)) .

Therefore, when the formulae at the bottom of page 4 --without the Q inserted --have been proved, our only additional proof obligation is:

P0 and Q => wp(S1, wp(S2, Q))

With respect to our original program we say that we have "proved the invariance of Q".

Consider now two programs, operating on the same variables. Suppose further, that with respect to each program we have proved the invariance of the assertions occurring in the other (or: occurring in the others, when we have three or more of such programs). This is, of course, a very strong assumption. But if it is satisfied, we have proved something useful about the following non-deterministic implementation.

Let us start a machine in an initial state satisfying each program's initial assertion. We now allow the execution of an arbitrary one of the programs to proceed until its next assertion. Firstly we have proved that this assertion

will then hold, secondly we have proved that the initial assertion(s) of the other program(s) have not been disturbed. Then, again, an arbitrary program is allowed to proceed with its execution until the next assertion, etc. When all programs have finished, all final assertions will hold.

Mind you: we are not talking about concurrency yet. We are talking about a nondeterministic machine, that can take care of the progress of a bunch of sequential programs, and we have stated conditions under which we can certainly allow a certain degree of interleaved execution, viz. from assertion to assertion.

As the reader will have noticed, I have mentioned a few times "suppose that we have proved proper termination". I made that caveat, because we would like to apply our theory also to a bunch of programs with the property that for the individual programs proper termination cannot be proved. The termination of a repetitive construct in the one program may depend on the execution of the other program having reached a certain stage. This will certainly be the case when we implement synchronization constraints by means of a busy form of waiting. In a case like that, we cannot even "prove" the termination of the bunch of programs without further assumptions about the daemon that makes the choice how to interleave: the bunch would not terminate if every time the daemon selected the waiting process to perform the next inspection of the unchanged state of affairs! The fact that a proof of termination of the whole bunch may require assumptions about the friendliness of the daemon justifies postponement of that issue.

It is not only the repetitive construct, for which the daemon's degree of being tamed can be an issue, also the alternative construct might, if we so desire, call for a certain amount of friendliness of the daemon. It could, for instance, be one of the daemon's restrictions, that an alternative construct, preceded immediately by its "preceding predicate" will never be selected for execution in those machine states where its selection for execution would lead to abortion of that program.

For the time being we assume that there is at least one sequence of choices by the daemon that will lead to proper termination of all the programs, and we assume the daemon to be friendly enough to choose such a sequence.

But even for that target, our formalism has to be changed: we have to

replace the weakest pre-conditions wp(S, P) which guarantee proper termination in a final state satisfying P by the so-called "weakest liberal pre-conditions" wlp(S, P) guaranteeing that the mechanism S will not terminate in a state not satisfying P . (This is the transition from total correctness, where the production of the right result is guaranteed, to partial correctness, where only the production of a wrong result is excluded. C.A.R.Hoare has taken this step a long time ago, and apparently at that time without much hesitation; I don't like it too much and would not like to take it unless I felt forced to do so.)

* * *

The next step is to introduce the possibility of concurrent execution, but to do it in such a way that, firstly, it is easily implementable, and, secondly, that no further nondeterminacy is introduced. For this purpose we divide the variables over various classes. On the one hand we have the private variables; private variables are always private to a specific program, viz. the only program that is allowed to refer to them. They are the local variables of the program they are private to, the other programs cannot inspect their values, nor change them. On the other hand we have the so-called common or shared variables: they are the remaining variables, to which at least two processes refer. It is clear that all interaction between the different programs must take place via the shared variables.

Each program is executed from assertion to assertion; here we assume that evaluation of a guard from a guarded command set implies the evaluation of all the guards from that set. The step from each assertion to the (dynamically) next assertion --our considered grain of interleaving-- we call "a unit of action". We now impose upon our units of action the constraint that they can be implemented with at most one access to at most one shared variable. With a memory switch that, in case of competition, orders the individual accesses to memory in some way or another, it is now clear that we can allow concurrent execution of as many units of actions as we have still incompleted programs. The reason that we are allowed to do so is that, no matter how we mix them, there always exists an order in which the units of action, executed one at a time, would have established the same net effect. Two units of action referring to two different common variables (or to no common variables at all) commute, for two units of action referring to the same common variable we can take the order in which the switch has granted them access to that shared variable.

Our restriction as regards access to shared variables has severe consequences: the guards of a guarded command set may refer to at most one shared variable. On the other hand, we now know that, with B a shared variable

```
{P1} if B → S1
     [] non B → S2
   fi {P2}
```

will not lead to abortion. (Note, that in the case of two successive inspections of B it is hard to prevent that, when the first inspection has encountered the value false , the next inspection may encounter the value true .)
Note, that, if in the above example, B is not a common variable (nor an expression referring to one), the guards of the guarded command set do not refer to a shared variable, and that in that case S1 may refer once to a common variable, and S2 may refer once to a different common variable: we have two possible units of action! For the time being, this is about the only thing I intend to say about concurrency.

* * *

F Here two assertions are missing! Sorry!
END.

Consider now the two programs

```
{P0} in1:= true;
{P1} do in2 →F in1:= false;
                {P2} in1:= true {P1}
     od;
     luck1:= true;
{P3} critical section 1;
{P3} luck1, in1:= false, false;
{P4} noncritical section 1
```

PROGRAM 1

```
{Q0} in2:= true;
{Q1} do in1 →F in2:= false;
                {Q2} in2:= true {Q1}
     od;
     luck2:= true;
{Q3} critical section 2;
{Q3} luck2, in2:= false, false;
{Q4} noncritical section 2 .
```

PROGRAM 2

with P0: non luck1 , we can prove

P1: non luck1 and in1

P2: non luck1 and non in1

P3: in1 and luck1

P4: non luck1 and non in1

and similarly for the Q's in Program 2. Furthermore we observe that all the Pi imply P: luck1 => in1 , and, similarly, that all the Qi imply Q: luck2 => in2 .

We can now replace all the original assertions Pi in Program 1 by Pi and Qj for any j : the proofs remain valid, because Program 1 does not refer to the variables mentioned in Qj . Similarly we can replace all the original forms of Qi in the second program by Qi and Pj for any j : again the proofs remain valid, because Program 2 does not refer to the variables mentioned in Pj . Having thus proved that the assertions of each of the programs are invariant with respect to the other program, we can conclude the universal validity of P and Q .

Finally we consider the relation R : non(luck 1 and luck 2) . Also this relation can be added to all assertions, it is also everywhere valid. The critical assignment in Program 1 that could destroy its validity is, of course "luck1:= true", but it is safe, because

$$wp(\text{"luck1:= true"}, R) = \text{non luck2} \quad ,$$

a condition that is implied by Q and non in2 . We interpret the universal validity of R as the guarantee of mutual exclusion in time of the two critical sections.

* * *

The classical use of critical sections has been the maintenance of an invariant relation

$$IR(a, b, c)$$

between a number of shared variables --here denoted by a , b , c-- , where this invariance cannot be maintained by a single unit of action, as a result of which a modification of the variables a, b, and c always implies a temporary violation of IR(a, b, c), after which it is again restored. With the aid of the additional variables we can replace it by a relation which is, indeed, universally valid, viz.:

$$\text{luck1 or luck 2 or } IR(a, b, c) \ .$$

Under the assumption that the pieces of program denoted by "noncritical sections" do not refer to the shared variables a, b, and c --nor to the private variables "luck", of course-- the proof that the noncritical sections leave this relation invariant is trivial. For the critical sections --the only pieces of program that are allowed to refer to a , b , and c-- it suffices to give the invariance proof for each of the critical sections in isolation.

At the beginning of critical section 1 --i.e. immediately after the assignment "luck1:= true" , we can assert

$$\text{luck1 and } IR(a, b, c) \quad . \qquad (1)$$

Internally, within the critical section 1 , we can introduce, wherever $IR(a, b, c)$ is temporarily violated, assertions of the type

$$\text{luck1 and } IR'(a, b, c, priv1) \qquad (2)$$

where with "priv1" we have denoted any other variables --besides luck1-- that are private to Program 1. At the end of the critical section 1 --i.e. just before luck1 is reset to false-- we must have again assertion (1) . We assume a similar proof that critical section 2 , considered in isolation, as a whole does not violate $IR(a, b, c)$.

The reasons why these two separate proofs for the critical sections in isolation suffice, is that assertions (1) and (2) are invariant with respect to Program 2 (and vice versa). The internal statements of critical section 2 cannot violate them, because their preceding predicates all contain the factor "luck 2", and the universal validity of R:

$$\text{non}(\text{luck1 and luck2})$$

ensures that the conjunction of these predicates and the assertions (1) and (2) is F ; because false implies everything, these proofs of invariance are trivial. The statements in noncritical section 2 cannot violate them either, because they don't refer to the variables occurring in (1) or (2).

Note. These proofs are so trivial that within critical sections the constraint that what we consider as "units of actions" refer at most to one shared variable can be weakened. Because, with a private variable "register"

$$\text{register:= c; } \{\text{register = c}\}\ \text{c:= register + 1}$$

gives rise to an internal assertion "register = c" which is trivially invariant, it is tempting to consider then the alternative c:= c + 1 as a unit of action. Such shortcuts should only be introduced with creat care. (End of note.)

* * *

Our solution for the mutual exclusion problem uses essentially two shared variables in1 and in2 . (They are really the only two variables that matter: the variables luck1 and luck2 are so-called "ghost variables" which have only been introduced for the sake of being able to formulate what we mean by "mutual exclusion" and of being able to formulate the proofs. In the actual

programs to be executed they --and all operations operating on them-- can be eliminated.) We also know that this solution is not acceptable when we reject solutions with the danger of after-you-after-you blocking. This danger is exorcized by Dekker's solution, which I give below in the following form. The initial value of the shared integer "turn" should be either 1 or 2 . I only give Program 1 ; Program 2 can be obtained from it by interchanging 1's and 2's.

```
{P0} in1:= true;
{P1} if in2 → {P2} if turn = 1 → skip {P3}
                  [] turn ≠ 1 → {P4} in1:= false;
                                {P5} do turn ≠ 1 → skip {P5} od;
                                {P6} in1:= true {P3}
                fi;
             {P3} do in2 → skip {P3} od
       [] non in2 → skip
    fi;
    luck1:= true;
{P7} critical section 1;
{P7} turn:= 2;
{P7} luck1, in1 := false, false;
{P8} noncritical section 1
```

Studying this program in relative isolation, we derive, under the assumption further

P0: non luck1

P1: non luck1 and in1

P2: P1

P3: non luck1 and in1 and turn = 1

P4: non luck1

P5: non luck1 and non in1

P6: non luck1 and non in1 and turn = 1

P7: luck1 and in1

P8: non luk1 and non in1

Again the relation luck1⇒ in1 is implied by all of them, and together with Program 2 we can derive the universal validity of non(luck1 and luck2) as before.

The difference between this program and the program on page 8 is that we need only weaker assumptions about the daemon if we would like to be sure of termination of the program on page 11. With the program on page 8, the daemon could select an unbounded number of units of actions from Program 1 and an unbounded number of units of actions from Program 2, without ever one of the critical sections being selected. With our new programs this is no longer true.

Selection of an infinite number of units of actions from program 1 implies --because there are only two loops in it, and from at least one an infinite number must be selected-- the validity of

(P5 and turn ≠ 1) or (P3 and in2)

or (non in1 and turn ≠ 1) or (in1 and in2 and turn = 1) (3)

(Note that the term "turn = 1" in the Pi is invariant with respect to Program 2.) For Program 2 we have the corresponding relation

(non in2 and turn ≠ 2) or (in1 and in2 and turn = 2) (4)

The conjunction of (3) and (4) reduces to

(non in1 and non in2 and turn ≠ 1 and turn ≠ 2) .

And, indeed, when we start the two programs with, say, turn = 3, the infinite looping of both programs is quite easily realized. If, however, we start the two programs -- and so we assume-- with

turn = 1 or turn = 2 (5)

then it is easily seen that (5) is invariant with respect to both programs, therefore can be regarded as universaly valid, and thus implying the falsity of the conjunction of (3) and (4). This falsity is usually taken as the proof of the absence of the danger of after-you-after-you blocking (and, a fortiori, the absence of the danger of deadlock).

The conclusion that the machine executing the programs' units of action in interleaved fashion will eventually terminate, rests on the assumption that the daemon will not be so grossly unfair as to select always the next unit of action from the same program. From a formal point of view this is a most unattracti assumption.

It would introduce a mechanism of unbounded nondeterminacy, it would give us means for implementing

"set x to any positive integer"

without being able to give an upper bound for the final value of x . We could, for instance, replace in program 1 the statement do in2 → skip od by

x:= 1; do in2 → x:= x + 1 od .

The consequences of introducing unbounded non-determinacy are sufficiently horrifying to reject the above approach.

Such a little loop with a skip as the repeatable statement is, of course, too indirect a way of indicating that to all intents and purposes, this program should not continue. We supply it with a kind of "fake continuation". The only way of not making assumptions about the fairness of the daemon is to restrict it explicitly in its freedom. The alternative construct gives us a way out.

In normal sequential programming we have regarded an alternative construct with all its guards false as a reason for abortion. An equivalent rule for the implementation would be: postpone progress of this computation as long as all the guards are false. In a uniprogramming environment we have "once all false, always all false" and this second rule would be as good as abortion. In a multiprogramming environment it would mean for the daemon that, as suggested on page EWD554 - 6,"an alternative construct, preceded immediately by its "preceding predicate" will never be selected for execution in those machine states where its selection for execution would lead to abortion of that program". By replacing in the program on page EWD554 - 11

do turn ≠ 1 → skip od by if turn = 1 → skip fi

and

do in2 → skip od by if non in2 → skip fi

and postulating that the daemon will not select a unit of action that starts with an alternative construct with false guards only, we have eliminated from this example all unbounded repetitions. To what extent the ideal "no unbounded repetitions in the individual programs" can be achieved in general --possibly by allowing certain special units of action to refer to more than one shared variable-- is a question to which I don't know the answer at the moment of writing.

14th of March 1976
Burroughs, Plataanstraat 5
NUENEN - 4565, The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow