Copyright Notice

Exercises in making programs robust.

(This is a sequel to the very exploratory EWD452: "About robustness and the like" which was initiated in September 1974 and closed on 31st January 1975.)

In this report I shall pursue a very simple idea. Provided that we give an adequate formulation of what we admit as "a single machine malfunctioning", we can interpret the effort as that of making a program in such a way that under the assumption of at most a single malfunctioning, the machine will never produce a wrong result as if it were the right one. I shall not, however, start my considerations with a very precise definition of the class of malfunctionings I am going to allow a single instance of: the probability that I have designed a tool of which, after much hard labour, we must conclude that it is insufficient for reaching our goal, is then just too high. I shall therefore start at the other end, and investigate the consequences of applying a technique which --with a certain amount of goodwill-- can be viewed as "making" a program more robust" and afterwards analyse, which class of malfunctionings it catches under the assumption of at most a single instance. The more elaborate exercises, I am sorry to announce, will be rather painful ones, because we cannot do them with too simple examples: if the example is very simple --like forming the sum of a hundred stored values-- the only way to make the program more robust boils (in some way or another) down to doing the computation twice and I am --obviously!-- more interested in what we can achieve without paying that price. (All by itself, this observation is already somewhat alarming: under assumption of a perfect machine, we are used to break down the whole computation as a succesion of little steps, all of them trivial by themselves, but if they can only be made more robust by duplication, our robustness concerns force us to consider larger "units". This seems a warning, that we are tackling a nasty subject!)

* * *

A very simple example to start with. A common program structure to establish a relation R is

(1) establish P; do BB → S od

where (P and BB) => wp(S, P)

and (P and non BB) => R

and we could replace (1) by

(2) establish P; do BB → S od; if P and non BB → skip fi

where the added statement causes abortion if the loop terminates with non P or BB , i.e. in a state in which we are not entitled to conclude the validity of R.

Time-wise this seems an attractive modification, because it does not generate an overhead on the repeatable statement S . An example would be (for $N \geq 0$) with

R: $a^2 \leq N$ and $(a+1)^2 > N$ and P: $a^2 \leq N$

to add to the program

a:= 0 {P}; do $(a+1)^2 \leq N$ → a:= a+1 od {R}

the checking statement

if $a^2 \leq N$ and $(a+1)^2 > N$ → skip fi

But this example immediately illustrates the very restricted --i.e. nearly empty-- range of applicability of this transformation: it only works in those cases where finding the answer may be hard, but checking the answer is (always!) easy. These cases seem to be rather the exception than the rule, and it would not amaze me if, often, when we think that we have found an example, the property that the correctness of a result is so easily checked can be used to speed up the process of finding one. (The above square root example is, indeed, ridiculously inefficient for larger values of N.)

* * *

What do we do, if --as for instance, when the correctness proof appeals to the Linear Search Theorem-- the verification of P and non BB amounts to redoing the computation? Very crudely, if our first program operates on a variable (set) x

(3) establish P(x);
do BB(x) → S(x) od

we could introduce a second set of variables, y say, and duplicate (under the assumption of determinacy)

(4) establish P(x);
do BB(x) → S(x) od;
establish P(y);
do BB(y) → S(y) od;
if x = y → skip fi

We can also merge the two processes, but

establish P(x) and P(y);
do BB(x) → S(x); S(y) od;
if x = y → skip fi

is a little bit too optimistic if we allow --and I think that we should-- erroneous sequencing as would result from an erroneous evaluation of a guard as possible malfunctioning:

(5) establish P(x) and P(y);
do BB(x) → if BB(y) → S(y) fi; S(x) od;
if non BB(y) and x = y → skip fi

is in this sense safe.

Up till now, there has been no gain by the transition from (4) to (5). But a fairly common structure of type (3), however, operates on a state space (x, z) and has the general form

establish P1(z) and P2(x, z);
do B1(z) and B2(x, z) → x:= f(x, z); z:= g(z) od

Here, repeated application of z:= g(z) generates a sequence of z-values --on account of B1(z) possibly finite-- and in the variable x , some function value of this sequence of z-values is computed (collected, if you prefer). The relation P1(z) --which z:= g(z) will keep invariant-- has been introduced to represent any possible redundancy in the representation of z. (If this redundancy is absent, P1(z) does not depend on z at all, is identically true and the remainder of this section --probably the whole report-- is no longer applicable.) If B2(x, z) is identically true, the sequencing is independent of x and, therefore, of the function f. If however, we are looking for the first z-value (if any) that satisfies some property --e.g. if we are looking for the smallest divisor less than the square root plus one--

B2 indicates that the search can be stopped as soon as a z-value satisfying the criterion has been found.

Again, we can merge to two copies, but what about letting the two state spaces share the same z ?

(6)
```
establish P1(z) and P2(x, z) and P2(y, z);
do B1(z) and B2(x, z) →
      if B1(z) and B2(y, z) → y:= f(y, z) fi;
      x:= f(x, z); z:= g(z)
od;
if non (B1(z) and B2(y, z)) and x = y → skip fi
```

How good is (6)? Suppose that the values of x , y and z are currently all correct, but that the evaluation of a guard is incorrect: as this incorrect evaluation is supposed to be the only malfunctioning, it will either itself cause abortion, or the next guard evaluation will do so. Suppose that the value of x has been corrupted and that this was our only malfunctioning, which is assumed to imply that y and z are and will remain correct. There are three cases: either we will, during $x \neq y$, encounter a case that $B2(x, z) \neq B2(y, z)$ and this will cause abortion; the second possibility is that, although $x \neq y$ remains, this will not occur, but then the last guard will cause abortion (on account of $x = y$). The third possibility is that this last abortion will not occur, because in the mean time $x = y$ has been re-established, i.e. the (apparently information destroying) operation $x:= f(x, z)$ has absorbed the malfunctioning: apparently, it did not matter! For a corruption of y (with the assumption that then x and z are, therefore, correct) the same applies. We are left with a corruption of z.

The operation $z:= g(z)$ is already supposed to satisfy

(7) $(P1(z)$ and $B1(z)) \Rightarrow wp("z:= g(z)", P1(z))$,

i.e. it is supposed not to destroy the validity of $P1(z)$. If we assume that the operation $z:= g(z)$ will, in addition, not destroy the validity of non $P1(z)$

(8) $(\text{non } P1(z)$ and $B1(z)) \Rightarrow wp("z:= g(z)", \text{non } P1(z))$

--i.e. will keep $P1(z)$ invariant in the strict sense--, then changing the last line of (6) into

```
if non (B1(z) and B2(y, z)) and x = y and P1(z) → skip fi
```

will guarantee that a corruption of z will be caught as well, if we assume that

(9) z is represented in such a redundant fashion, that any corruption of it, that would not destroy the validity of $P1(z)$ can be regarded as a multiple malfunctioning, or, to put it in another way, each single malfunctioning affecting z will make $P1(z)$ false.

* * *

I have done extensive exercises with a program solving the following problem: generate all cyclic arrangements of 16 zeroes and 16 ones, such that all 32 possible configurations of 5 successive bits occur (and therefore: exactly once). Another formulation of describing the same problem is: generate all permatations $h_0 \ldots h_{31}$ of the numbers 0 through 31 that

0) $h_0 = 0$,

2) $suc(h_i, h_{i+1})$ for $0 \leq i < 31$

3) $suc(h_{31}, h_0)$,

where $suc(a, b) = (a \ \underline{mod}\ 16 = b \ \underline{div}\ 2)$.

It is in the latter form that we shall tackle it. First of all: because $suc(0, x)$ has only the solutions $x = 0$ and $x = 1$, and $h_1 \neq h_0$, it follows that $h_1 = 1$. Therefore in a permutation satisfying 1) and 2) it follows that $h_{31} = 16$, for: $suc(16, x)$ has as only solutions $x = 0$ and $x = 1$ and thus, for all $i < 31$ we have $h_i \neq 16$. In short, we can drop for the permutation requirement 3) as it is implied by the others. The original inner block as designed by W.H.J.Feijen, was essentially the following one:

```
begin virvar x; privar h, p;
   x vir int array := (0); h vir int array := (0, 0);
   p vir bool array := (0, true); do p.dom ≠ 32 → p:hiext(false) od;
   do h(0) = 0 →
        begin glovar x, h, p; privar c;
           if h.dom < 32 → skip
            [] h.dom = 32 →
                 begin glovar x; glocon h; privar j;
                    j vir int := 0;
                    do j ≠ 32 → x:hiext(h(j)); j:= j + 1 od
                 end
           fi;
           c vir int := 2 *(h.high mod 16);
           do p(c) →
                 do odd(c) → c, h:hipop; p:(c)= false od;
                 c:= c + 1
           od;
           h:hiext(c); p:(c)= true
        end
   od
end
```

The extensive exercises, however, have been thrown into the waste-paper basket, because they had a very ad hoc character and the proofs that the resulting programs were resistant to a single malfunctioning either failed or became so laborious as to become unconvincing. It was that disappointing experience that prompted me to try to formulate --"in abstracto" so to speak-- what I was really doing, while designing the above robust structure (6). My next experiment will therefore be to try the above general technique in a hopefully systematic manner to this specific program. (In order to keep the experiment fair, I shall not exploit the fact that something more about the answer is known: it has been proved that the number of solutions equals 2048 , but we continue as if this theorem were unknown to us.)

To establish the connection: the general x of (6) corresponds to the output array x of our example and the role of the general z of (6) has been taken over by the pair h, p in our example. Relation P2 is the simple (and not too interesting) one: in our example it is

P2(x, h, p): the value of the array variable x "consists" of all solutions in alphabetical order that precede alphabetically the permutations that begin with

$$h(0) \ldots h.high \quad .$$

(The term "consists" is loose, but hopefully clear enough. It is further to be noted that in the above formulation of P2 , the boolean "presence" array p is not mentioned.)

Relation P1 is more interesting: it consists of two terms P1.1(h) and P1.2(h, p):

P1.1(h): for all i satisfying $h.lob \leq i < h.hib$ we have $suc(h(i), h(i+1))$

P1.2(h, p): for all k satisfying $0 \leq k < 32$,
p(k) implies that there exists 1 value for i, and
non p(k) implies that there exists no value for i , such that

$$h.lob \leq i \leq h.hib \text{ and } h(i) = k \quad .$$

According to P1.1 , the integer array h contains in general redundant information: a boolean array --manipulating the bits of the original statement of the problem would have done the job also--. Feijen replaced the boolean array by an integer array for reasons of efficiency.

According to P1.2 the boolean presence array p stores purely additional information that follows functionally from h; it has been introduced also for reasons of efficiency by Feijen.

And here lies our hope for gain: the redundancy that we need for the robust presentation of z may already be present for efficiency's sake!

We may wonder, whether the redundancy provided by h and p is sufficient: because p follows uniquely from h , a scrambling of the value of p will always violate P1.2 . It is, however possible to scramble h without violating P1.1 nor P1.2 (it is difficult, but it can be done). This can be remedied by replacing the boolean "presence" array p by an integer "place" array p, satisfying the new

P1.2(h, p): for all k satisfying $0 \leq k < 32$,
either $p(k) = -1$ and there exists no value i satisfying $h(i) = k$,
or $0 \leq p(k) \leq h.hib$ and then $i = p(k)$ is the only value for $i \geq 0$ (see below), satisfying $h(i) = k$.

As the cost is negligible and it is our plan to do a thorough job, I propose to switch to the integer "place" array p. (The last requirement $i \geq 0$ has been added because it is a simplification to extend the array h at the low end with $h(-1) = 16$ for the verification of P1.1 : upon removal of a top element the array h does not become empty.)

The critical operation is now "z:= g(z)". We must change it so as to satisfy (8) as well. We can, indeed, insert additional tests that would lead to abortion if the intended modification of z would lead to a violation of non P1(z) , but this is not sufficient, because how do we know that the

correct new value of z has been assigned to it? (If z:= g(z) would have acted as a skip, we might produce the same solution twice!)

The critical value, of course, is that of "c": if the initialization of c had erroneously been carried out as

c vir int := 2 *(h.high mod 16) + 1 ,

a whole class of solutions could be skipped.

So we had better concentrate upon the active scope of c and repeat our games (or similar ones: wait and see). We have for the active scope of c --i.e. more precisely: until the extension h:hiext(c)-- if all goes well the invariant relation

P3(h, c): suc(h.high, c) .

Because (non P3(h, c)) => wp("h:hiext(c)", non P1.1(h))

it suffices, as far as our obligations versus the invariance of non P1.1(h) is concerned, to keep during the active scope, besides non P3(h, c) also non(P1.1(h) and P3(h, c)) invariant, i.e. we do not need to check, whether

c, h:hipop

perhaps could destroy non P1.1(h), because that would imply the emergence of non P3(h, c), which will not disappear unnoticed.

As P3(h, c) covers the four most significant digits of c, the least significant digit of c seems to be our remaining Achilles heel. I propose to count the number of even numbers among

h(0) through h(h.hib), extended with c during the latter's active scope

This will catch erroneous initialization of c ; if the guard odd(c) is erroneously evaluated, an even c will disappear without the count being decreased, if the gaurd is erroneously evaluated false, c:= c + 1 will increase the number of even values, while it should decrease them by one. This count is a kind of fancy parity bit. The full program is shown on the next p

(Warning: the proof reading of the program text on page EWD482 - 6 has not been done with the same care as I spent to the pages of my book.)

Let me give some explanatory notes.
The outer guard h(0) = 0 is not repeated automatically, if true: it only matters, when we think that we have found a solution, and then it should be confirmed by p(0) = 0; this means that after the last solution has been found and p(1) is already = 1, it would not be detected if the outer repetition were on for a while. Why should it?

The operations, which are essentially of the form x:= f(x, z) and y:= f(y, z) are themselves fully unchecked: if something goes wrong there that is harmful, different values of x and y will result. Note that the test, whether a new solution has been found is repeated: once for x and once for y .

The conclusion that p(c) $\geq$ 0 holds, has to be confirmed, otherwise the erroneous conclusion that extension with c would lead to duplication would cause possibly a large collection of solutions to be skipped. (This additional confirmation was lacking in my first version of the robust program

```
begin virvar x, y; privar h, p, n; n vir int := 1;
   x vir int array := (0); y vir int array := (0); h vir int array := (-1, 16, 0)
   p vir int array := (0, 0); do p.dom ≠ 32 → p:hiext(-1) od;
   do h(0) = 0 →
     begin glovar x, y, h, p, n; privar c;
        if h.dom < 32 → skip
         [] h.dom = 32 → if p(0) = 0 → skip fi;
           begin glovar x; glocon h; privar j; j vir int := 0;
              do j ≠ 32 → x:hiext(h(j)); j:= j + 1 od
           end
        fi;
        if h.dom < 32 → skip
         [] h.dom = 32 → if p(0) = 0 → skip fi;
           begin glovar y; glocon h; privar j; j vir int := 0;
              do j ≠ 32 → y:hiext(h(j)); j:= j + 1 od
           end
        fi;
        c vir int := 2 *(h.high mod 16); n:= n + 1;
        do p(c) ≥ 0 → if p(c) ≥ 0 → skip fi;
              do odd(c) → if suc(h.high, c) → c, h:hipop fi;
                          if p(c) = h.hib + 1 → p:(c)= - 1 fi
              od;
              c:= c + 1; n:= n - 1
        od;
        if suc(h.high, c) → h:hiext(c) fi;
        if p(c) = - 1 → p:(c)= h.hib fi
     end
   od;
   if h.dom = 2 and h(0) = 1 and n = 0 and p(0) = - 1 and p(1) = 0 →
      begin glocon p; privar j; j vir int := 2;
        do p(j) = - 1 and j < 31 → j:= j + 1 od;
        if p(j) = - 1 → skip fi
      end
   fi
end
```

The comparison of the global values x and y , which should be equal, has been delegated to the surroundings.

I observed the omission while typing these notes!) The conclusion that on account of non $p(c) \geq 0$ the repetition has to be terminated is asked for confirmation 7 lines lower.

The test $odd(c)$ in the innermost repetition does not need further confirmation, as all erroneous evaluation would leave its traces in a non-correct value of n.

Finally, at the end of our original program, it is checked --somewhat superfluously-- that $h.dom = 2$; the test $h(0) = 1$ is necessary for the confirmation that the outermost repetition has not stopped too early, thereby possibly missing a number of the last solutions. Finally $P1.2(h, p)$ is fully checked. (We can regard the test $h.dom = 2$ as part of that test, so perhaps its presence is fully justified after all.)

And this concludes my treatment of this example.

* * *

As the plurals in my title betray, I had originally in mind to deal wit more examples. At second thought I shall confine myself in this report to thi single example: I am already on the eighth page, with single space typing. Although I had announced, that the exercises would be "rather painful ones", I did not expect, that it would be so painful. So I think that I should send the report now away, as it stands, hoping for helpful comments. Therefore a f concluding remarks.

If the inefficiency of our final program "hurts", we should be aware of the following considerations. Why does it "hurt"? Well, because the many test that we have inserted, are on the one hand assumed to absorb computer time, on the other hand --unless the machine is completely lousy-- will be very skew: of course, for if the machine were perfect, they would not give information at all! The normal reaction to such very skew tests has been to devote dedicated hardware to them (vide the parity check or the interrupt circuit). If techniques, as displayed in this report, would be applied to general purpo programs --note, that I have not made up my mind, whether that would be a goo thing!-- this conflict could perhaps be solved by the presence of some program-controlled hardware that could do some of the checking in parallel with the main computation.

For the time being, techniques as shown are probably more appropriate in special purpose environments, such as, for instance, micro-programs or just the instruction cycle. One of the reasons for undertaking all this was my growing doubt, whether our techniques for the quality control of both, chip design and chip construction, is sufficient. If techniques like the abov can be transferred to that more microscopic level, we might feel confident to catch in a single stroke both design errors and incidental machine malfunctionings.

20th March 1975
Plataanstraat 5
NUENEN - 4565
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow