A little exercise in deriving multiprograms

by W.H.J. Feijen

In this note we record an experiment in deriving multiprograms from their functional specifications, with the predicate calculus and the theory of Owicki and Gries as our only tools for reasoning. For the benefit of the experiment we have selected an example problem that is so simple that it need not divert our attention from the subject matter, which is the process of derivation.

Someone who is familiar with the theory of Owicki and Gries may, right at the outset, be amazed about our choice to use that theory for the purpose of deriving multiprograms. Since, after all,

(i) isn't that theory too simple to deal with something as complicated as parallel programs?

(ii) and isn't it the case that that theory addresses partial correctness only, thus completely ignoring the important issues of deadlock and individual starvation?

(iii) and hasn't that theory been designed just for aposteriori verification of multiprograms?

We think that these questions are legitimate, and we therefore wish to spend a word on them.

As for (iii), we have to bear in mind that the theory of Owicki and Gries emerged in a period when computing scientists had just begun to explore the possibility of deriving -sequential- programs. In those days the formal derivation of multiprograms was not within the scope of immediate interest, and definitely beyond the then technical competence. We mention the latter so explicitly because meanwhile we learned that the possibility to derive programs formally cannot come without the willingness to abandon interpretative reasoning. In the mid 70s, operational understanding of programs and interpretation of mathematical formulae still were the predominant options. In particular, computing scientists still had to develop the predicate calculus and by now we know that without its mastery, program derivation can hardly be done. With this in mind, it is quite understandable why the theory of Owicki and Gries has received the stigma of having been designed just for aposteriori verification: it has always been used for that.

As for (ii), the only thing we can do for the moment is to refer the reader to our forthcoming example. It gives us the

opportunity to derive a whole spectrum of solutions to the programming problem, with at the one end of the spectrum a solution that displays a wealth of potential parallelism and doesn't suffer from the danger of starvation, and with at the other end of the spectrum a solution that displays deadlock and, hence, no parallelism at all. Drawing from limited experience we can say that the spectrum is always there and that the "good" half of it remains within reach, provided one adheres to a design discipline in which one doesn't commit oneself too easily to premature decisions.

As for (i), what else can we say than that the most effective weapon against lurking mathematical complexity is a simple formalism to begin with. And as yet, we have no indication that the theory of Owicki and Gries could be too naive.

* * *

The theory of Owicki and Gries can be briefly explained as follows. A multiprogram is a set of sequential programs, that may be annotated with assertions. The theory tells us that we can then rely on the correctness of the annotation whenever for each assertion we can show

- that it is established by the program in which it occurs, the so-called local correctness of the assertion,

- and that it is maintained by the atomic statements of the other programs, the so-called global correctness of the assertion.

In principle this is it. Note that in using the theory, we will always have to be very explicit about which are the atomic statements.

In dealing with multiprograms we use weakest liberal preconditions for the characterization of statements. The most noticeable difference with weakest preconditions is that we now have for the alternative construct (with one guard)

$$\text{wlp.}(\underline{\text{if}}\ B \rightarrow S\ \underline{\text{fi}}).R \equiv \neg B \vee \text{wlp.}S.R \quad .$$

It entitles us to conclude the local correctness of assertion B in the program fragment

$$\underline{\text{if}}\ B \rightarrow \text{skip}\ \underline{\text{fi}}\ \{B\} \quad .$$

and this, in fact, is the most important thing we need to know for what follows.

Near the very end of our example derivation we will also use a program transformation, the validity of which is captured by the following lemma that we mention without proof. The purpose of the transformation is to reduce the grain of atomicity, thus enhancing the degree of potential parallelism, but without impairing the program's correctness.

Lemma We consider replacing an atomic alternative construct

if $B \wedge C \rightarrow$ skip fi

in one of the component programs with the sequence

if $B \rightarrow$ skip fi
; if $C \rightarrow$ skip fi

of two atomic alternative constructs. The lemma is, that the replacement is harmless to the correctness of the annotation, provided B is not falsified by any atomic statement of any other component program. Moreover, the replacement does not introduce the danger of deadlock. (End of Lemma.)

And after these preliminaries, we are ready for our example derivation.

* * *

We consider a terminating multiprogram in which, at the outset, each component program consists of a single assignment to a local boolean variable:

Prog.i : [[$y.i$: bool ; $y.i$:= $B.i$]],

for some boolean expression $B.i$ still to be determined. The problem is to synchronize the components in such a way

that the final state of the multiprogram satisfies

(0) $\quad (\underline{N}i :: y.i) = 1,$

i.e., precisely one of the booleans y has the value true. The synchronization has to be realized by means of atomic statements of the traditional type "at most one access to at most one shared variable". Furthermore the synchronization has to be carried out so as to meet the following "fairness" requirement.

Relation (0), viewed as an equation in y, has as many solutions as there are component programs, and the fairness requirement is that our ultimate multiprogram can generate each solution of (0). Since we do not allow this requirement to penetrate our discussion, we immediately satisfy it by deciding that our design be symmetric in the component programs.

So much for the problem statement.

Our design process - and this is typical - comprises two stages. In the first stage we design a program annotation so that its assumed correctness implies the synchronization condition - (0) in our case - . In the second stage we use the theory of Owicki and Gries to realize the correctness of the annotation. We begin with the first stage.

* * *

Given the assignment $y.i := \mathcal{B}.i$, we cannot assert much more than that Program i establishes postcondition $y.i \equiv \mathcal{B}.i$, and we cannot hope for much more than that the other programs will maintain it. So we decide that

$y.i \equiv \mathcal{B}.i$ is a correct postcondition in Program i,

with expression $\mathcal{B}$ still to be determined.

Expression $\mathcal{B}$ has to follow from what we now know, viz. that the multiprogram establishes postcondition

$$(\underline{A}i :: y.i \equiv \mathcal{B}.i) ,$$

and from what we have to guarantee, viz. that it establishes postcondition (0). We can now satisfy (0), provided we can design $\mathcal{B}$ such that the postcondition satisfies

$$(\underline{N}i :: \mathcal{B}.i) = 1 ,$$

or - equivalently -

(1) $(\underline{E}i :: \mathcal{B}.i)$ and

(2) $(\underline{A}i,j :: \mathcal{B}.i \wedge \mathcal{B}.j \Rightarrow i=j)$.

And this is precisely what we shall do.

Choice false for $\mathcal{B}$ evidently meets (2), and it is the only choice that does so without taking the implication's consequent into account. But the choice is

no good for (1), so that we had better consider that consequent. It mentions an equality sign, and here we have to remember that the only way to conclude the equality of two arbitrary expressions is by using that equality is transitive. Thus we arrive at our next (and last) choice for B, viz.

$$B.i \equiv v = i ,$$

for some v. Having satisfied (1), we are left with the obligation to ensure that the postcondition of the multiprogram satisfies (1).

With our choice for B, Program i can establish (1) by an assignment $v := i$ to a fresh shared variable v. In view of the desired symmetry between the component programs there is hardly any other possibility either.

And here we have reached the end of the first stage. Summarizing, we achieved that the multiprogram with shared variable v and with components given by

Prog.i:
$$\begin{array}{l} [\![\ \ldots \\ \ ; \ v := i \\ \ \ \ldots \\ \ ; \ y.i := \ v = i \\ \ \ \ \{ \ y.i \equiv \ v = i \ \} \\ ]\!] , \end{array}$$

meets target relation (0), provided the assertion $y.i \equiv v = i$ in Program i is correct. It is this proviso which we shall realize in the second stage of our design.

* * *

Intermezzo

In showing the global correctness of an assertion P in one of the programs, we have to show that for each atomic statement S in a different program,

$$P \Rightarrow wlp.S.P$$

is a theorem. But what if it isn't? The answer is that in that case the annotation of the programs is not strong enough. It is standard practice then to strengthen the annotation by adding a new assertion C, say, as a conjunct to P, and by adding a new assertion D, say, as a conjunct to the precondition of S, and to design the newly introduced assertions in such a way that they are correct and satisfy

$$C \wedge D \Rightarrow (P \Rightarrow wlp.S.P) .$$

Note that for given P and S this implication yields an equation in the unknown predicates C and D, and that it has at least one solution, viz. for $C \wedge D \equiv false$.

(End of Intermezzo.)

Now we address the required correctness of assertion $y.i \equiv v = i$ in Program i. Its local correctness is obvious. For its global correctness we observe that the only statements from other programs that may falsify it are the assignments to v, for which we have to guarantee that they don't. I.e. we have to ensure that for any j, $j \neq i$,

$$(y.i \equiv v = i) \Rightarrow \text{wlp.}(v := j).(y.i \equiv v = i),$$

or – equivalently, using the axiom of assignment and $j \neq i$ –

$$(y.i \equiv v = i) \Rightarrow \neg y.i,$$

or – equivalently, using predicate calculus –

(3) $$\neg y.i \vee v \neq i .$$

This, however, is hardly a theorem and we therefore strengthen the annotation as indicated in the above Intermezzo:

Prog.i:
$$\begin{array}{l} [\![\ \ldots \\ \quad \{D.i\} \\ ; \ v := i \\ \quad \ldots \\ ; \ y.i := v = i \\ \quad \{C.i\} \ \{ y.i \equiv v = i \} \\ ]\!] \ . \end{array}$$

(Juxtaposition of assertions denotes their conjunction.) Now the global correctness of assertion $y.i \equiv v = i$ is guaranteed, provided we can design C and D in such a way that

(4a) Assertions $C.i$ and $D.i$ are correct ;

(4b) $(\underline{A}j : j \neq i : C.i \wedge D.j \Rightarrow (3))$.

And this is precisely what we shall do.

As yet we cannot do much with requirement (4a), but (4b) enables us to eliminate $C.i$; by (3) and predicate calculus, (4b) is equivalent to

$$C.i \Rightarrow (\underline{A}j : j \neq i : \neg D.j \vee \neg y.i \vee v \neq i).$$

We "strengthen" this requirement on $C.i$ a little bit by removing the disjunct $\neg y.i$, yielding

$$C.i \Rightarrow (\underline{A}j : j \neq i : \neg D.j \vee v \neq i).$$

(This "strengthening" is done in view of the fact that $C.i$ is a postassertion of an assignment to $y.i$; anticipating that we still have to ensure $C.i$'s local correctness, it is attractive to have $C.i$'s solution space as independent of $y.i$ as possible. Moreover the "strengthening" is no real strengthening, since assertion $C.i$ occurs in conjunction with the already established assertion $y.i \equiv v = i$.)
We now have an abundance of choices for $C.i$, for instance

(a) false
(b) $(\underline{A}j : j \neq i : v \neq i)$
(c) $(\underline{A}j : j \neq i : \neg D.j)$
(d) $(\underline{A}j : j \neq i : \neg D.j \vee v \neq i)$,

but we will select the weakest one, i.e

we choose

$$C.i: \quad (\underline{A}j : j \neq i : \neg D.j \ \vee \ \nu \neq i) \ .$$

The more pragmatic reason for this choice is that the weaker we choose our assertions the less we constrain the potential parallelism of our ultimate solution. The more fundamental reason is that we can always strengthen an assertion later on, should the need arise.

Having settled (4b), we are left with the task of ensuring (4a). Assertion $C.i$ is so wildly different from any other assertion in Program i that there is hardly any other way of ensuring its local correctness than by "testing" it. Thus we obtain

```
Prog.i:   [[ ...
            {D.i}
          ; ν := i
            ...
          ; if C.i → skip fi
            ...
          ; y.i := (ν = i)
            {C.i} { y.i ≡ ν = i }
          ]] .
```

By incorporating the D's in the guard, we have committed ourselves Fa representation Fto of the predicates D by, say, a bunch of fresh shared boolean variables, one per program. In view of assertion $D.i$, their initial values had better be true.

The global correctness of $C.i$ offers no

problem at all. The assignments $v := j$, for $j \neq i$, just make $C.i$ more true. In this respect, we could also easily accommodate assignments $D.j := \text{false}$, which are absent now ... .

The latter remark is of course related to the observation that the multiprogram as we have it now, is a pretty naive one. Without assignments $D := \text{false}$, we can simplify $C.i$ into the rejected alternative (b), and we can eliminate the D's altogether. The resulting program suffers from the danger of starvation, and that is not what we are after. So let us accommodate assignments $D := \text{false}$.

By the place of occurrence of assertion $D.i$ in Program i, the assignment $D.i := \text{false}$ has to succeed $v := i$, and as far as the correctness of the annotations D is concerned there are no other constraints on where to plug it in. For reasons of "maximal progress", however, we plug it in so that the guards become true "as soon as possible", i.e. immediately following $v := i$. Thus we arrive at what, apart from a final transformation, will be our ultimate program. Fully encoded and completely annotated, it is

Initially: $(\underline{A}i :: D.i)$

Prog.i: $[\![$ $\{D.i\}$
$v := i$
; $D.i := \text{false}$
; $\underline{\text{if}}\ (\underline{A}j : j \neq i : \neg D.j \vee v \neq i) \rightarrow \text{skip}\ \underline{\text{fi}}$
; $y.i := v = i$
$\{(\underline{A}j : j \neq i : \neg D.j \vee v \neq i)\}\ \{y.i \equiv v = i\}$
$]\!]$
$\{(\underline{N}i :: y.i) = 1\}$.

We omit the proof that there is no danger of deadlock or starvation.

The final transformation concerns the breaking up of the rather coarse-grained alternative construct into a sequence of fine-grained ones. Here we can use the Lemma mentioned in the beginning of this note. Since the guard is a finite conjunction and since none of the conjuncts $\neg D.j \vee v \neq i$ is falsified outside Program i, the evaluation of the conjunction can be carried out conjunct-wise, even in any order. Finally, each atomic statement $\underline{\text{if}}\ \neg D.j \vee v \neq i \rightarrow \text{skip}\ \underline{\text{fi}}$ is of a type that can be implemented by "at most one access to at most one shared variable".

And this concludes the derivation of the algorithm.

* * *

We may wonder how the potential parallelism would have been constrained had we chosen a stronger $C.i$ than

we did. With e.g. choice (c), we would have obtained all but the same multiprogram, except that the alternative construct would have been **if** (∀j: j≠i: ¬D.jj) → skip **fi**. The resulting algorithm would have been a "two-phase algorithm", in which first all component programs perform their assignment to v and when they have all done so then their assignments to y. In our current algorithm no such "system-wide synchronization point" exists: the two phases are sweetly interleaved. With the still stronger choice (a) for C.i, we would have derived programs containing **if** false → skip **fi**. This would indeed cut down the potential further parallelism, even dramatically so, because the multiprogram would be guaranteed to come to a premature halt.

Eindhoven, November 1969