10. Concurrent Vector Writing:
a first exercise in program development

In the foregoing chapters we have seen a number of examples of how to use the technique of Strengthening the Annotation to show that a given, too weakly annotated multiprogram is correct. There we strengthened the original annotation in a number of steps, until it became correct in the Core. Here, in this chapter, we will do exactly the same, be it not for a completed program, but for a program yet to be completed. That is, along with a stronger annotation, new, additional code will see the light as well, and that is what we call program development. We have to admit, though, that in this first exercise not too much additional code will be developed, on the one hand because the example is very simple, and on the other hand because a cautiously carried out development does not introduce more than needed.

Besides using the example to give a first performance of program development, we will also use it to explain more precisely our bookkeeping regime with the "Notes" and the "queries", which we have been using a number of times now without much ado. We will also be more explicit about the status of the inter-

mediate versions that emerge during the approximation process, which starts at the specification and ends in the ultimate solution.

* * *

The problem of Concurrent Vector Writing is the following. We consider an array $x[0..N)$, $0 \leq N$, and two components A and B. Component A writes zeroes into x, and component B ones. We wish to synchronize the components such that the multiprogram terminates and delivers a final state satisfying $\vec{x} = \vec{0}$.

A more precise specification of this problem is given by the figure below.

Pre:	$i = 0 \wedge j = 0$
A:	**do** $i \neq N \rightarrow x.i := 0 \; ; \; i := i+1$ **od**
B:	**do** $j \neq N \rightarrow x.j := 1 \; ; \; j := j+1$ **od**
Post:	$? \langle \forall k : 0 \leq k < N : x.k = 0 \rangle$

Version 0

(The individual assignments are atomic.)

This specification is more precise in that it reveals <u>how</u> the components write into x. We shall refer to the combination of the precondition Pre and the program texts for A and B

as the *Computation Proper*.

Remark The legal status of the computation proper is that it is owned by someone else, who needs it for some purpose that is not of our concern. This implies that no matter how we proceed, we are not entitled to interfere with this computation in any other way than needed for the purpose of synchronization.
End .

Version 0 also contains a notational device that we shall use all through, namely the query. A query indicates that something remains to be done —here, take care of the correctness of the postcondition—. As we go along, we will also encounter queried assertions and queried system invariants (like we did in earlier examples).

Taking care of a queried item means accomplishing its correctness (in the Core). This will always be achieved by extending the program text —be it with code, assertions, or invariants—. However, these extensions are constrained by the rule that

the computation proper is not to be changed

This rule prohibits, for instance, that we replace the parallel composition of A and B with the

sequential program B;A , which would vacuously establish the required postcondition. This would not do: the legal proprietor has handed in a parallel program.
The rule also prohibits that we plug in further changes to x, i, or j. Inspections, however, of the variables of the computation proper will, in general, be allowed — sometimes they are even unavoidable (for instance in termination detection algorithms) — . In many circumstances, though, the additional code will be phrased in terms of fresh (synchronization) variables.

Given the above conventions regarding the status of the computation proper and given our appreciation of queried items, Version 0 acts as the formal specification of the programming task ahead of us.

Remark Admittedly, the repertoire of atomic statements that we may draw from to solve the problem has been left unspecified; in particular, nothing has been said about the grain of atomicity in the guards and in the assignments. Our general pursuit is for quite fine-grained solutions, but we do not want to commit ourselves in too early a stage. We want to tackle the logic of the design first and see what kind of expressions emerge in the additional code. Only then will we be con-

cerned with "implementation issues" like making solutions more fine-grained -if necessary and technically feasible- .
End of Remark.

* * *

The simplest solution to our current problem is one that mimics the behaviour of the sequential program B; A. This can be achieved at the expense of one fresh boolean, f say. A solution is

```
Pre:    i = 0 ∧ j = 0 ∧ ¬f

A:  if f → skip fi          B:  do j ≠ N →
  ; do i ≠ N →                      x.j := 1
       x.i := 0                   ; j := j + 1
     ; i := i + 1             ; od
    od                          f := true
```

It is a nice exercise to prove that this program, indeed, establishes the required postcondition.

Although there is no logical objection to this solution, there is a strategical one: all the potential parallelism has been killed. Component A starts idling for f to become true, i.e. it is denied progress until B has terminated, whereas there seems to be no good reason for this delay.

This is not quite proper (towards the legal owner of the computation proper), and therefore we will always try to obey the rule that

> the computation proper is
> not to be hampered (= delayed)
> without good reasons

Sometimes there are good reasons for hampering the progress of the computation proper, and near the end of this chapter we will encounter one such reason, which is of a rather quantitative nature. It stands to reason, however, that the best reason for hampering progress is when otherwise the required synchronization, i.e. the (partial) correctness of the design, would be endangered. Safety first!

* * *

After these preliminaries, we now proceed with our problem by making a first design decision, namely the decision to restrict our solution to a multiprogram of the form

Pre: $i=0 \wedge j=0$	
A: **do** $i \neq N$ $\rightarrow$ S_0 ; $x.i := 0$; S_1 ; $i := i+1$; S_2 **od**	B: **do** $j \neq N$ $\rightarrow$ T_0 ; $x.j := 1$; T_1 ; $j := j+1$; T_2 **od**
Post: ? $\langle \forall k : 0 \leq k < N : x.k = 0 \rangle$	

Version 1

The S's and the T's serve as *placeholders* for the additional code: they indicate the only places where we (have decided to) allow synchronization code to be inserted. With the above choice of placeholders we have been quite generous, but not completely so: we have ruled out the "sequential" solution suggested earlier, by offering no placeholders before and after the repetitions.

Remark It is, in general, a good habit to be explicit in the matter of placeholders, and in our current example we are -and will be until the very end-. Drawing from experience, however, we know that they are mostly well-understood, so that we can afford to leave them implicit. But

when in doubt, we had better not leave them out.

In the rest of this monograph, we will only occasionally feel the need to introduce place-holders, and fortunately so because, in fact, they are a notational mistake. It would be far more efficient and elegant to have a notational device for indicating where no additional code may be inserted, but we have not been able to design a notation that would be plain enough to be adopted and put into practical use.

End of Remark.

* * *

Version 1 contains one queried item, viz. the postcondition. From the Rule for the Postcondition we know that its correctness is guaranteed whenever it follows from the conjunction of the postconditions of the individual components. Now observe that

$$\langle \forall k : 0 \leq k < N : x.k = 0 \rangle$$
$$\Leftarrow$$
$$\langle \forall k : 0 \leq k < i : x.k = 0 \rangle \ \wedge \ i = N \ .$$

From the structure of component A we see that $i = N$ is a correct postcondition of A, variable i being private to A. So what remains is the correctness of

P: $\langle \forall k : 0 \leq k < i : x.k = 0 \rangle$

as a postcondition of A or B. We enforce it – and this is a design decision – by demanding that P be a system invariant. With these choices it suffices to take true as a postcondition of B, and because true is a correct assertion anywhere in a multiprogram, we never write it down.

Thus we arrive at our next version.

Pre: $i=0 \wedge j=0$	
Inv: $?P$: $\langle \forall k : 0 \leq k < i : x.k = 0 \rangle$	
A: **do** $i \neq N$ → S_0 ; $x.i := 0$; S_1 ; $i := i+1$; S_2 **od** $\{i = N\}$	B: **do** $j \neq N$ → T_0 ; $x.j := 1$; T_1 ; $j := j+1$; T_2 **od**
Post: $\langle \forall k : 0 \leq k < N : x.k = 0 \rangle$	

Version 2

Observe that the postcondition has lost its query, because in passing from Version 1 to Version 2

we turned it into a correct one, be it at the expense of a -here vacuously correct- assertion $i = N$ in A and a new queried item, viz. system invariant P.

This is the place to draw attention to what we think is an important methodological issue. Version2 tells us in a very precise and compact way what has been achieved -viz. the correctness of assertion $i = N$ and the postcondition- and what remains to be achieved -viz. the correctness of the queried items, here system invariant P-. Thus the figure named Version 2 acts as the precise interface between the past and the future of the development process. As a result, Version2 is the specification of the programming problem ahead of us, and it is absolutely irrelevant how we arrived at it. This is important because, as the development evolves from one version to the next, we at any time only need to be concerned with how to transform the current version into a next one. Thus the procedure is very similar to the method of stepwise refinement (and, in a way, very similar to calculation, where, at any time, we are -to a large extent- only concerned with how to transform the current expression into a next one.)

This, too, is the place to argue once more why

such a transformation is correct in the light of the Owicki-Gries theory. In transforming Version 1 into Version 2, we have clearly strengthened the annotation: the latter contains assertion $i = N$ and -queried- system invariant P, which were both absent from the former. Now suppose that in the end we succeed in removing the queries from Version 2 by appropriate choices for the S's and the T's; then Version 2 will have turned into a correctly annotated multiprogram. But thanks to the postulate of Weakening the Annotation, Version 1 will have turned into a correctly annotated multiprogram as well, and that is what we were after!

* * *

After these intermediate remarks, time has come to resume our development. Version 2 requires that we take care of the invariance of P. To that end we have to investigate whether P holds initially and under what additional conditions it is maintained by the atomic statements of the multiprogram. We record these investigations as follows:

Re Inv P "$\langle \forall k: 0 \leq k < i : x.k = 0 \rangle$"

Init: correct, from $i = 0$ in Pre

• $\{true\}$ $x.i := 0$

- $\{\ ?\ x.i = 0\}\quad i := i + 1$
- $\{\ ?\ i \leq j\ \}\quad x.j := 1$

End .

Because the situation is so simple, we have given the additional preconditions at once, i.e. without giving the calculations leading to those preconditions. Observe that we tacitly used the Rule of Orthogonality by ignoring all statements that could not possibly affect P; these are $j := j+1$ and the S's and the T's, which were supposed to not change the computation proper, i.e. to not change x, i, or j.

Remark During class-room sessions on this example, students are asked what precondition ought to be supplied to $x.j := 1$ in order that P not be violated. Quite a few come up with the answer $i<j$, and that is correct. It is just a tiny little bit stronger than our $i \leq j$. However minor this difference may seem, it has major consequences: with condition $i<j$, (individual) deadlock will become unavoidable. We urge the reader to check this after he has studied the rest of this treatment. The moral is that multiprograms are unusually delicate artefacts and that their design is a highly critical activity. When in our chapter on Strengthening the Annotation we said that we always wanted the weakest additional con-

dition, we meant just that. The consequence is that well-versedness in the predicate calculus is indispensable for playing this game, and there is no escaping it.
End of Remark .

The incorporation of these new pre-assertions leads us to the next version. It reads

Pre: $i=0 \wedge j=0$	
Inv: $P: \langle \forall k : 0 \le k < i : x.k = 0 \rangle$	
A: do $i \neq N$ → S_0 ; $x.i := 0$; S_1 ; $\{ ? \; x.i = 0 \}$ $i := i+1$; S_2 od $\{ i = N \}$	B: do $j \neq N$ → T_0 ; $\{ ? \; i \le j, \text{Note0} \}$; $x.j := 1$; T_1 ; $j := j+1$; T_2 od
Post: $\langle \forall k : 0 \le k < N : x.k = 0 \rangle$	

Version 3

In moving to the next version we have to remove one or more queries. In general, the choice of how many and which ones to remove is

completely free. We indicate the ones chosen by supplying them with a reference to a Note. Thus, Version 3 expresses our intention to tackle $i \leq j$ first, leaving $x.i = 0$ for later.

Note 0 "$i \leq j$"

We ensure the correctness of $i \leq j$ by requiring it to be a system invariant.

End of Note 0.

Why this design decision, which looks much stronger than necessary? The alternative would have been to ensure the local correctness of $i \leq j$ in B through guarded skip if $i \leq j \rightarrow$ skip fi, but this would not accord with the Ground Rule for Progress: component A has no potential for effectively weakening $i \leq j$. Hence the "choice" of letting $i \leq j$ be a system invariant is more or less imposed on us.

Remark In the current example, the adoption of guarded skip if $i \leq j \rightarrow$ skip fi would nevertheless have led to a happy end, but that is just a stroke of very good luck. We would like to encourage the reader to trace this alternative derivation after he has studied this one.
End of Remark.

Now we should write down the next version,

with " ? i≤j " appearing under the heading "Inv", but in order to shorten our treatment a little bit, we skip this intermediate version and deal with the invariance of $i \leq j$ right away. We will apply this form of *cascading* more often.

Re Inv $i \leq j$.

Init: correct, from Pre

• $\{ ? \; i<j \}$ $i := i+1$

End .

Thus we arrive at

Pre: $i = 0 \wedge j = 0$
Inv: P: $\langle \forall k: 0 \leq k < i : x.k = 0 \rangle$, $i \leq j$
A: do $i \neq N$ → S_0 ; $x.i := 0$; S_1 ; $\{ ? \; x.i = 0$, Note$\}$ $\{ ? \; i<j$, Note 1$\}$ $i := i+1$; S_2 od $\{ i = N \}$

```
B:   do  j ≠ N
     →   T0
         ; {i ≤ j}
           x.j := 1
         ; T1
         ; j := j + 1
         ; T2
     od
```

Post: $\langle \forall k : 0 \le k < N : x.k = 0 \rangle$

Version 4

We continue with the remaining obligations.

Note 0 "$x.i = 0$", with co-assertion $i < j$

L: Choose $S1$ = skip. Then $x.i = 0$ follows from the textually preceding $x.i := 0$.

G: Only $x.j := 1$ in $\mathcal{B}$ can violate $x.i = 0$. We calculate

$$
\begin{aligned}
& (x.j := 1) \,.\, (x.i = 0) \\
\equiv \quad & \{ \text{substitution} \} \\
& x.i = 0 \;\wedge\; i \neq j \\
\equiv \quad & \{ i < j \text{ is a co-assertion of } x.i = 0 \} \\
& x.i = 0 \quad .
\end{aligned}
$$

Hence, $x.i = 0$ is not violated, thanks to its co-assertion $i<j$ —see Remark below—.

End of Note 0.

Remark From our earlier chapter on Strengthening the Annotation we recall that in showing the global correctness of an assertion we are allowed to use its co-assertions, and that is what happens in the above. Nevertheless, the reader might feel somewhat uneasy about using a co-assertion —$i<j$— that still carries a query. But here we should not forget that in the end such a query will be removed —here turning $i<j$ into a correct assertion.
End of Remark.

Note 1 "$i<j$", with co-assertion $x.i = 0$

L: Because we have chosen S_1 equal to skip (see Note 0), we ensure the local correctness of $i<j$ by demanding

$$\{ ?\ i<j\ ,\ \text{Note 2} \}\ x.i := 0\ .$$

G: Widening.

End of Note 1.

Note 2 "$i<j$"

L: Choose guarded skip $\underline{\text{if}}\ i<j \rightarrow \text{skip}\ \underline{\text{fi}}$ for S_0.

G: Widening.

End of Note 2.

Observe that, again, we used cascading, by handling the new assertion $i<j$ emerging in Note1.L right away.

Now we are done: all queries have been removed! Before writing down our solution, however, we have to admit that we played a tricky game with the order in which the assertions $x.i = 0$ and $i<j$ in Version4 were tackled. Had we first dealt with $i<j$ and decided to establish its local correctness by choosing if $i<j \rightarrow$ skip fi for S1, the design would have become much more complicated. (The reason is that $i<j$ is really needed for the global correctness of $x.i = 0$.) We invite the reader to try this alternative.

The final solution, in which the remaining placeholders are now omitted — or rather: replaced by skips — is as follows

Pre:	$i = 0 \wedge j = 0$
Inv:	P: $\langle \forall k : 0 \leq k < i : x.k = 0 \rangle$, $i \leq j$
A:	**do** $i \neq N \rightarrow$ **if** $i < j \rightarrow$ skip **fi** ; $\{ i < j \}$ $x.i := 0$; $\{ x.i = 0 \} \{ i < j \}$ $i := i + 1$ **od**
B:	**do** $j \neq N \rightarrow$ $\{ i \leq j \}$ $x.j := 1$; $j := j + 1$ **od**
Post:	$\langle \forall k : 0 \leq k < N : x.k = 0 \rangle$

Version 5

Our only remaining task is to show that the multiprogram terminates. Component B surely does, thus establishing $j = N$. Then, guard $i < j$ of A's guarded skip is stably true, because $i < N$ is a correct precondition of the guarded skip.

Hence, A terminates as well.

* * *

Herewith we conclude the formal development of an -admittedly- very simple multiprogram. Our main purpose was

- to show in very small steps how such a development evolves
- to show how the validity of such a development is carried by the theory of Owicki and Gries
- to exhibit the similarity with "stepwise refinement" and its induced benefits: the stimulation of a better separation of concerns
- to present some of the notational and clerical aids used in organizing such a development.

A few final remarks are in order

• Again, our final version is fully documented in that its annotation is correct in the Core. But this need no longer amaze us, because it is intrinsic to the game. Do observe, however, how crisp the annotation is; here we are reaping the benefits of a cautious development of the program: nothing is encountered that is not strictly needed.

• In our presentation we proceeded from one version to the next in very small steps. We did so in order to explain the rules of the game. Sometimes, when the situation had become simple and transparent, we accelerated the design process a little by what we called "cascading", thus combining several successive versions into one. Such a combination of steps certainly shortens the presentation, but it may also evoke oversights or mistakes. It is always a matter of good taste how coarse- or how fine-grained one chooses one's steps. In this respect we have one rule of thumb that we ourselves learned from sad experience: in case one is on the verge of making mistakes or loosing one's grip, the advice is: Slow Down.

• Presenting a development like this on the blackboard is much simpler and faster, because with a little orchestration and preparation there will be no need to copy the successive versions, as is necessitated here since paper is such a linear medium. On the blackboard we can begin by writing down the computation proper, leaving conspicuous space for the placeholders and for the assertions, and then fill up the space with code and assertions as the development evolves. As for the queries: instead of erasing a query from its item, we can extend it with its mirror image, thus forming symbol ♡, to indicate that the item has become

sound. Because in this way program texts can only grow, one never needs a brush. This may be nice to know for the teaching reader.

* * *

And this concludes our first program derivation.

Postscript (for the circuit designer)

In our solution – Version5 –, the synchronization runs under control of the originally given program variables i and j. In fact, only the difference j-i matters – see the guarded skip in component A –. In the algorithm, this difference is an (N+1)-valued entity, because $0 \leq j-i \leq N$ is a system invariant. For circuit design, it could be advantageous if j-i were just 2-valued, because that would enable a transformation into the boolean domain. For this (good) reason, we shall now constrain the computation proper by strengthening system invariant $i \leq j$ to

Q: $\quad i \leq j \;\wedge\; j \leq i+1$.

This has an impact on the code of component B, which increments j. We give the adjusted B

at once, leaving to the reader to check that relation Q has, indeed, become a system invariant. We also insert two obviously correct assertions for later usage.

Pre: $i = 0 \wedge j = 0$	
Inv: Q: $i \leq j \wedge j \leq i + 1$	
A: **do** $i \neq N \rightarrow$ **if** $i < j \rightarrow$ skip **fi** ; $x.i := 0$; $\{i < j\}$ $i := i + 1$ **od**	B: **do** $j \neq N \rightarrow$ **if** $j \leq i \rightarrow$ skip **fi** ; $x.j := 1$; $\{j \leq i\}$ $j := j + 1$ **od**

For reasons to become clear shortly, B has been given the same syntactic structure as A has.

By inserting a guarded skip in B, we may have hampered progress, so the question is: does this program still terminate? It does, because

- at least one of the guards of the two guarded skips is true, so that
- $i+j$ increases, and
- since $i \leq j \leq N$ is a system invariant, B will terminate in a state satisfying $j = N$, so that

- as before, A will terminate as well.

Now we transform the program according to the coordinate transformation

$$c \equiv (i=j) \quad , \text{ or - equivalently, see Q -}$$
$$\neg c \equiv (i+1=j) \; .$$

Then we can eliminate the guards thanks to

$$i<j \equiv \neg c \qquad \text{and} \qquad j \leq i \equiv c \; .$$

Statement $\{i<j\}\ i := i+1$ can now be replaced with

$$\{\neg c\}\ c, i := \neg c,\ i+1$$

or - using pre-assertion $\neg c$ - by

$$c, i := \text{true},\ i+1 \; .$$

The raw program code thus becomes

Pre: $i=0 \wedge j=0 \wedge c$	
A: do $i \neq N$ → if $\neg c$ → skip fi ; x.i := 0 ; c, i := true, i+1 od	B: do $j \neq N$ → if c → skip fi x.j := 1 c, j := false, j+1 od

* * *

And now the situation has become so symmetric that if we want the above machinery to deliver $\vec{x} = \vec{1}$ instead of $\vec{x} = \vec{0}$, we only need to flip c's initial value. And if we want $\vec{x} = \vec{0}$ or $\vec{x} = \vec{1}$, i.e. if we don't care, we simply leave c's initial value unspecified. It is with this regime that two individual processors can write a bit stream, bit by bit, into one and the same memory location, such that the result is as if they had done so in some unspecified order.

* * *

We conclude this postscript with two final remarks.

- We will discuss coordinate transformations like the one applied above with more precision in a later chapter.

- We achieved the two-valuedness of j-i by strengthening the annotation via introduction of invariant Q, thus reducing the degree of parallelism that can be displayed by the program. We will encounter this trade-off more often and then discuss it at greater length.

9 March 1997
W.H.J. Feijen and A.J.M. van Gasteren