A Formal Development of Distributed Summation

Rob R. Hoogerwoord

11 april 2000

Contents

Chapter 0

Introduction

Chi va piano va sano.

0.0 why formal reasoning?

Formal correctness proofs serve several purposes. First, a formal proof may be needed for the purpose of *a-posteriori* (and possibly mechanical) verification of an already existing program. Second, a formal proof may provide a guideline for the construction of a program, thus giving rise to "correctness by design". Somewhere in between these two, a formal derivation of an already existing program may contribute to a better understanding of the mathematical structure of its design. This entails providing answers to questions like: what mathematical properties are needed and where in the proof are they needed, and what design decisions have been taken and were these really necessary?

Although this is an a-posteriori activity too, its emphasis is not on verification but on understanding. It is based on the observation that, however useful formal verification is, the mere fact that *somebody else*, or even some machine, has verified a proof does not make *me* understand it any better. This understanding, in turn, is important if we wish to be able to design larger programs and still be able to guarantee their correctness.

More specifically, by making design decisions explicit, alternative choices become visible as well, and possibly variations of, if not improvements upon, the algorithm at hand may be identified. Thus, instead of a single algorithm actually a whole class of similar algorithms can be studied. For example, in all treatments of "Distributed Summation" I have seen [2, 3, 7], the use of a rooted spanning tree is taken for granted; our development shows that there is no logical necessity for such trees: a connected, acyclic, directed subgraph is what is needed, and a spanning tree is (just) one of the options.

Another important aspect is the possibility to clearly separate problem-

specific from general-mathematical knowledge needed in the proof: those steps in the proof where problem-specific knowledge is used are the steps that distinguish the problem at hand from other problems; often these are also the steps where design decisions are taken.

In particular, the relation between a programming notation – a formalism – and its implementation – in terms of "execution traces" – is not specific for any particular programming problem, but can be fixed once and for all for that programming notation. Once the interface between notation and implementation has been established by means of proof rules, (correctness of) any particular program can be discussed in terms of these proof rules only.

In this setting, "formal" means "constructed according to accurately formulated rules" ; these rules are not necessarily the rules of traditional formal logic. We call a proof "formal" if each step in it can be justified by an appeal to one of the "accurately formulated rules" . For our (practical) purposes, these rules are the rules of predicate calculus and of whatever well-understood area of mathematics we need. (For example: sets, bags, relations, functions, partial orders, mathematical induction, the natural numbers and the integers are considered well-understood.)

0.1 the problem of distributed summation

In this study we present a formal development of an algorithm for the problem known as "Distributed Summation" . Informally, the problem is to design a distributed algorithm for calculating the sum of a collection of numbers; these numbers are distributed over the *nodes* of a given finite *network*, in such a way that every node holds one number. The computation is to be initiated by one dedicated node, called the *root*, and upon completion of the computation the sum of all numbers is to be available in the root. The algorithm must be "truly distributed" . This means that in every node in the network, no other information about the network is available than the identities of and the communication links with that node's direct neighbours. As a result, the component of the algorithm to be executed by a given node can only be based on that (very local) information.

The problem of "Distributed Summation" is of interest, because, first, it is about the simplest example of a distributed computation and yet complicated enough, and, second, it occurs as a subproblem in solutions to many other problems, such as distributed mutual exclusion, broadcast protocols, routing protocols, and the like. Note that "summation" and "number" must not be taken too literally here: the solution is applicable to any algebraic structure

with a symmetric and associative binary operator; for example: booleans with disjunction, conjunction, or equivalence, and sets or bags with union.

0.2 how we solve it

To a very large extent, distribution can be considered an implementation aspect: a distributed algorithm is just a parallel algorithm that happens to be implemented in a distributed way. If communication in the parallel program is represented by *shared variables*[0], and if every shared variable is shared by only two parallel components (of the program), then every shared variable can be implemented by means of a communication link between the two machines executing those two components.

If the number of available communication links in the underlying network is limited, the number of pair-wise shared variables has to be equally limited. Hence, the design of the algorithm will certainly be influenced by the requirements of a distributed implementation, but otherwise we can reason about distributed algorithms in very much the same way we reason about "ordinary" parallel programs.

In this study we use the Owicki-Gries formalism [6] to develop the solution and its proof of correctness. Its main advantages are the simplicity of its rules and the possibility to treat a program as a formal text in its own right; this enables us to reason about a program *without* reasoning about its execution traces.

For an extensive treatment of the Owicki-Gries formalism and its applications to program construction, we refer to [1]; here we confine ourselves to a short summary of the main notions and technical issues, *as used in this study*:

- A (parallel) program is the parallel composition of a number of *components*, where a component is a sequential program fragment, constructed from (so-called) *atomic statements*.
- Variables occurring in at least two components are called *shared* variables, and variables occurring in only one component are called *private*. Here "occurring in" pertains to assignments and expressions as well as assertions and invariants.
- Properties of the program are formulated as pre- and postconditions of the program or of its components, and as system invariants.

[0] not to be confused with shared *memory*!

- During the development of a program, additional invariants, assertions and statements in the program's components may (have to) be introduced.
- The precondition of the program must imply all preconditions of the components, and the postconditions of the components in conjunction must imply the postcondition of the program.
- Every assertion in every component must be *locally correct*, which means that it must be a valid postassertion of the immediately preceding atomic statement.
- Every assertion in every component must be *globally correct*, which means that it is not violated by any atomic statement in any other component. In such a proof the preassertion of that atomic statement may be exploited; assertion Q is not violated by statement (with precondition) $\{ P \}\ S$ means: $\{ Q \wedge P \}\ S \{ Q \}$.
- A (system) invariant is a predicate that is implied by the precondition of the program and that is not violated by any of the atomic statements in any of the components; as a result we may exploit an invariant as a valid assertion at any place in the program.
- A proposition $\{ Q \wedge P \}\ S \{ Q \}$ is trivially true if the variables occurring in Q are not modified in S. Whenever we prove global correctness of an assertion Q, we exploit this tacitly, by confining our attention to those statements in other components that do modify variables in Q. As a special case, if Q contains no shared variables at all, the requirement of global correctness of Q is void.
- A proposition $\{ Q \wedge P \}\ S \{ Q \}$ is trivially true whenever Q and P are disjoint, that is, $[Q \wedge P \Rightarrow \mathsf{false}]$. This is called the *rule of disjointness* and it is an important technical device to achieve global correctness: particularly if S, all by itself, would violate Q, the only option is to strengthen Q or P (or both) so as to obtain disjointness. (In operational terms, this is also known as *mutual exclusion*.)
- Some statements do not violate an assertion but, on the contrary, make the assertion only "more true"; this is called *widening*. Simple examples are: assertion b and statement $b := \mathsf{true}$, assertion $\neg b$ and statement $b := \mathsf{false}$, and assertion $x \leq y$ and statement $y := y+1$. Widening statements can often be identified easily and, subsequently, be ignored.

- *Synchronisation* is needed to establish local correctness of an assertion, if this cannot be achieved otherwise. By way of experiment, we use the notation $\{\bullet\, B\, \bullet\}$ – "guard B" – as an abbreviation of what could also be written as $\mathsf{await}\ B$, or as $\mathsf{if}\ B \rightarrow \mathsf{skip}\ \mathsf{fi}\,\{\, B\,\}$: thus, we avoid the operational connotations of the former and the elaborateness of the latter. The construct $\{\bullet\, B\, \bullet\}$ is both a statement and an assertion: as a statement, its execution requires B to be true and modifies no variables, and as an assertion, its local correctness is guaranteed. As a result, $\{\bullet\, B\, \bullet\}$ does not violate any other assertions, and the only proof obligation is the global correctness of the assertion B.

- The above only pertains to *partial correctness*, that is, without regard to *progress properties*, which must be proved separately. Fortunately, in many simple cases progress properties can be formulated as ordinary predicates on the state of the system and can thus be treated as safety properties. As we will see, distributed summation is such a simple case.

- Step-by-step proof construction is possible, thanks to a *monotonicity* property: the introduction of additional assertions or invariants does not violate the correctness of the already present assertions and invariants. In the same vein: strengthening a guard does not violate what already is correct.

0.3 how to read this document

We will develop the algorithm for distributed summation in a step-by-step fashion, dealing with one aspect of the problem at a time. As a result, a presentation emerges that displays a good separation of concerns (but that also is rather long).

To provide the reader with some assistance, we conclude every step with a summary of what has been achieved thus far. This summary is all that is needed to understand the steps to follow: it is the interface between two successive steps. Thus, the reader may (and even is advised to) skip some parts at first reading.

The development proceeds in a top-down fashion: the introduction of new notions and relations is postponed until they are really needed. This may require some patience on the part of the reader, but it is an essential technique to maintain clarity and to prevent premature, if not unnecessary, design decisions.

example: It is only by *not* taking the use of a spanning tree for granted and by *forcing* myself *not* to introduce it before the need would arise, that I was able to discover that we do not really need spanning trees at all.

□

0.4 notational conventions

Throughout the development, variables i, j, n, p, q, R have type *node*, where R is a constant, denoting the root node in the network. (Recall that the root is the node initiating the computation.)

We will introduce relations (on the set of nodes) named $\leftarrow$ and $\sim$. Informally, if $i \leftarrow j$ then we (sometimes) call i a *successor* of j and call j a *predecessor* of i. Relation $\sim$ will be symmetric, and we (sometimes) call i and j each other's *neighbours* if $i \sim j$.

For these relations we denote their *transitive closure* by $\overset{+}{\leftarrow}$ and $\overset{+}{\sim}$, respectively. Transitive closure can be defined in two, equivalent, ways. First, $\overset{+}{\leftarrow}$ is the *strongest* of all transitive relations $<$ (say), satisfying:

$$(\forall i, j :: i \leftarrow j \Rightarrow i < j) \quad ,$$

as a consequence of which we have both:

$$(\forall i, j :: i \leftarrow j \Rightarrow i \overset{+}{\leftarrow} j) \quad ,$$

and, for any transitive relation $<$:

$$(\forall i, j :: i \leftarrow j \Rightarrow i < j) \;\Rightarrow\; (\forall i, j :: i \overset{+}{\leftarrow} j \Rightarrow i < j) \quad .$$

Second, $\overset{+}{\leftarrow}$ can also be defined recursively by, for all i, j:

$$i \overset{+}{\leftarrow} j \quad \equiv \quad i \leftarrow j \;\vee\; (\exists p :: i \overset{+}{\leftarrow} p \wedge p \leftarrow j) \quad .$$

We will use both definitions just as we see fit; the proof of their equivalence belongs to the general-mathematical knowledge whose discussion falls outside the scope of this study.

For any predicate Q, not universally false and possibly containing variable p, we sometimes use an abstract statement $p : Q$ with the informal operational

meaning "assign to p such a value that Q holds"; formally, this means that Q is a (locally) correct postassertion of this construct. We use it whenever we do not wish to specify p more than that it should satisfy Q. For example: $p:\ p \leftarrow q$ means "select an (arbitrary) successor of q and assign it to p".

Chapter 1

The Summation Phase

1.0 specification

Every node j has an integer variable x_j, whose initial value is the number held by that node; we do not require that x_j retains its initial value: it is a true variable. In addition, the root node R has an integer variable y that, upon completion of the computation, is equal to the sum of the initial values of the variables x. For the sake of simplicity, we assume $y = 0$ initially.

To formalize this, we introduce a constant C as (a name for) the sum of the initial values of the variables x. So, the precondition of the algorithm is:

$$C = (\Sigma j :: x_j) \quad \wedge \quad y = 0 \quad ,$$

and its postcondition can now be formulated as:

$$C = y \quad .$$

A distributed algorithm for this problem will be the parallel composition of a number of *components*, one per node of the network. The informal requirement that "upon completion of the computation the sum of all numbers is to be available in the root" is properly reflected in the above postcondition, because y is a variable of the root. Nevertheless, this is not sufficient because how could, in a truly distributed system, the root possibly detect that the whole computation has terminated? So, we must strengthen the specification: we require $C = y$ to be a (local) postcondition of the root component. Thus we achieve that $C = y$ holds as soon as the root has terminated.

The algorithm should be such that, in its final implementation, all communication is restricted to neighbouring nodes in the (given) network, but this

requirement, which we call *network compliance* here, will enter the design in a rather late stage.

1.1 calculation of the sum

In our first approximation to the solution the correct answer will be calculated, without much regard to the requirements of a distributed implementation.

Were we heading for a traditional sequential program, a simple iteration of $y := y+x_q$, for all q, would do the job –given that y is zero initially–. A distributed implementation of this iteration is hard to envisage, however, because of the central role played by variable y.

Therefore, we investigate assignments of the shape $x_p := x_p+x_q$ (for judiciously chosen p and q). Because somehow y has to obtain a value, we also will need assignments –at least one– of the shape $y := \cdots$.

In addition we introduce a boolean variable b_j, one for every node j, to distinguish the variables that still contribute to the answer from the variables that do not anymore. Thus, we propose the following system invariant:

Q0: $\quad C \;=\; y \;+\; (\Sigma j : b_j : x_j) \quad .$

As usual, Q0 captures what the pre- and postconditions of the problem have in common. Q0 is implied by the precondition, provided initially variables b and y satisfy:

$$y \;=\; 0 \quad \wedge \quad (\forall j :: b_j) \quad .$$

The required postcondition, $C = y$, follows from Q0 if, in addition:

$$(\forall j :: \neg b_j) \quad .$$

So, the main purpose of the algorithm now is to set all booleans b to false, under invariance of Q0. To this end, each assignment to a b must be accompagnied by an assignment to one of the integer variables. We investigate this separately for the root node and for all other nodes.

The root R has $C = y$ as its postcondition, which follows from Q0 provided $(\forall j :: \neg b_j)$ is a valid postcondition of R as well. This, in turn, gives rise to $(\forall j : R \neq j : \neg b_j)$ as a necessary precondition of $b_R := \mathsf{false}$, which we therefore add as a guard. With this precondition and with b_R, we now have that $(\Sigma j : b_j : x_j)$ is equal to x_R, so the only thing we can do here is combine $b_R := \mathsf{false}$ with $y := y+x_R$. Thus, we obtain as component program for R:

$add{\cdot}R$: $\{\ b_R\ \}\,\{\bullet\ (\forall j : R{\neq}j : \neg b_j)\ \bullet\}$
$y\,, b_R\ :=\ y{+}x_R\,,\ \mathsf{false}$
$\{\ (\forall j :: \neg b_j)\ ,\ \text{hence: } C = y\ \}$

For any p and q satisfying $p{\neq}q \wedge b_p \wedge b_q$ we have:

$$(\Sigma j : b_j : x_j)\ \ =\ \ (\Sigma j : p{\neq}j \wedge q{\neq}j \wedge b_j : x_j)\ \ +\ \ (x_p{+}x_q)\quad ,$$

which, in turn, can be rewritten as:

$$(\Sigma j : b_j : x_j)\ \ =\ \ (\Sigma j : q{\neq}j \wedge b_j : x_j)\,(x_p := x_p{+}x_q)\quad .$$

This observation yields the following component for node q, for every q such that $R{\neq}q$; the global correctness of the assertion labelled with **??** remains to be established yet:

$add{\cdot}q$: $\{\ b_q\ \}$
$p\colon\ p{\neq}q \wedge b_p$
; $\{\ p{\neq}q \wedge b_p\ \mathbf{??}\ \}$
$x_p\,, b_q\ :=\ x_p{+}x_q\,, \mathsf{false}$
$\{\ \neg b_q\ \}$

aside: In all its simplicity, the above proposal represents quite a few design decisions; for instance, we could try to do without the booleans b and we could propose operations like $x_p, x_q := x_p{+}x_q, 0$, or even $x_p, x_q := x_p{+}h, x_q{-}h$, for any h. We shall not pursue this, but at the moment I have no evidence that such an approach would fail.

Our booleans b have been introduced to record more explicitly which of the variables x are still relevant; as is already visible in $add{\cdot}R$, they also play a role in the synchronization of the components.

□

We conclude this (and each next) subsection with a summary of the current approximation, so as to provide a clear interface between the successive steps in the development: this summary is all that matters for what is to follow, and all preceding considerations may now be forgotten.

The first approximation to the algorithm is the parallel composition of the components $add{\cdot}q$, for all q (including R).

summary 0 : (reminder: assertions between $\{\bullet\ \cdots\ \bullet\}$ are guards.)

pre: $C = (\Sigma j :: x_j) \ \wedge \ y = 0 \ \wedge \ (\forall j :: b_j)$

Q0: $C = y + (\Sigma j : b_j : x_j)$

post: $(\forall j :: \neg b_j)$

$add{\cdot}R$: $\{ \ b_R \ \} \{\bullet \ (\forall j : R{\neq}j : \neg b_j) \ \bullet\}$
$y, b_R := y{+}x_R$, false
$\{ \ (\forall j :: \neg b_j) \ ,$ hence: $C = y \ \}$

and for each q different from R:

$add{\cdot}q$: $\{ \ b_q \ \}$
$p: \ p{\neq}q \wedge b_p$
; $\{ \ p{\neq}q \wedge b_p$ **??** $\}$
$x_p, b_q := x_p{+}x_q$, false
$\{ \ \neg b_q \ \}$

□

1.2 a connecting graph

In view of the desire for a distributed implementation, the guard of $add{\cdot}R$ is awkward: it refers to all other boolean variables of the program, and thus it is much too global an expression. Rather, we prefer to formulate all guards in local terms only, where "local" means "involving a modest amount of shared variables only" .

So, we need a way to draw global conclusions from local assertions. For this purpose, we introduce a binary relation $\leftarrow$ on the set of nodes; for the time being, $\leftarrow$ is constant. We write $\overset{+}{\leftarrow}$ for the transitive closure of $\leftarrow$. In what follows, we will derive by careful analysis what properties $\leftarrow$ should have such that it serves our purposes well.

We begin with introducing an additional invariant, coupling relation $\leftarrow$ to the boolean variables b:

Q1: $(\forall i, j : i{\leftarrow}j : b_i \Leftarrow b_j \ )$.

The expression $b_i \Leftarrow b_j$ also defines a relation on the nodes, and this relation is transitive. Hence, from the definition of transitive closure, we obtain as a corollary of Q1 the following stronger property for free:

Q1+: $(\forall i, j : i{\overset{+}{\leftarrow}}j : b_i \Leftarrow b_j \ )$.

remark: This is why the transitive closure is useful: Q1 is formulated in "local" terms, namely pairs of nodes related by $\leftarrow$, and Q1+ is a stronger – "more global" – property, in terms of pairs related by $\overset{+}{\leftarrow}$.
□

Now we derive:

$$
\begin{array}{ll}
 & \text{Q1+} \\
\Rightarrow & \quad \{ \text{ instantiation } i := R \ \} \\
 & (\forall j : R \overset{+}{\leftarrow} j : b_R \Leftarrow b_j \,) \\
\equiv & \quad \{ \text{ predicate calculus } \} \\
 & b_R \ \vee \ (\forall j : R \overset{+}{\leftarrow} j : \neg b_j \,) \\
\Rightarrow & \quad \{ \text{ assuming R0, see below } \} \\
 & b_R \ \vee \ (\forall j : R \neq j : \neg b_j) \\
\equiv & \quad \{ \text{ predicate calculus } \} \\
 & b_R \ \vee \ (\forall j :: \neg b_j) \quad ,
\end{array}
$$

from which we conclude that, provided Q1 is invariant indeed, we have $\neg b_R \Rightarrow (\forall j :: \neg b_j)$. As a result, the guard of $add{\cdot}R$ can be removed, because its *raison d'être* has disappeared: the additional postcondition $(\forall j :: \neg b_j)$ of $add{\cdot}R$ is now implied by Q1 $\wedge \ \neg b_R$.

In the above derivation we have assumed that relation $\leftarrow$ satisfies the following requirement of (what we call) *connectivity*:

R0: $\quad (\forall j : R \neq j : R \overset{+}{\leftarrow} j \,) \quad .$

For the time being we just put R0 on our list of requirements imposed upon $\leftarrow$; these requirements will be dealt with in the connection phase.

* * *

A new invariant generates new proof obligations. In our case, Q1 is implied by the precondition of the whole program, and, both for $R{=}q$ and for $R{\neq}q$, the additional precondition, as required by Q1, for $add{\cdot}q$ is:

$$(\forall j : q \leftarrow j : \neg b_j) \quad .$$

We add this precondition as a guard to $add{\cdot}q$, and because it is stable, it also is a globally correct assertion. It is sufficiently "local", provided not too many j satisfy $q \leftarrow j$.

* * *

Invariant Q1 can also be used to establish the global correctness of the assertion $p{\neq}q \wedge b_p$, in $add{\cdot}q$: we now require p to satisfy $p{\leftarrow}q$, such that b_p follows from Q1 and b_q. If, in addition, $\leftarrow$ is *irreflexive* then $p{\leftarrow}q$ implies $p{\neq}q$ as well.

remark: We do not put irreflexivity on our list of requirements imposed upon $\leftarrow$, because it will follow from a stronger requirement that we will need later anyhow.
□

That a p satisfying $p{\leftarrow}q$ exists follows from the following little lemma.

lemma: $R{\neq}q \;\Rightarrow\; (\exists p :: p{\leftarrow}q)$

proof: From R0 $\wedge$ $R{\neq}q$, we conclude $R\overset{+}{\leftarrow}q$, and we derive:

$$
\begin{array}{ll}
 & R\overset{+}{\leftarrow}q \\
\equiv & \quad \{ \text{ transitive closure } \} \\
 & R{\leftarrow}q \;\vee\; (\exists p :: R\overset{+}{\leftarrow}p \wedge p{\leftarrow}q) \\
\Rightarrow & \quad \{ \text{ weakening } \} \\
 & R{\leftarrow}q \;\vee\; (\exists p :: p{\leftarrow}q) \\
\equiv & \quad \{ \text{ absorption } \} \\
 & (\exists p :: p{\leftarrow}q)
\end{array}
$$

□

As we have no particular reason to prefer one successor of q over the other, we deliberately formulate the choice of p nondeterministically here, thus effectively leaving it to the implementation.

Thus, we have arrived at our second, and final approximation of the summation phase, in which all assertions are correct now, albeit that we still must prove progress.

summary 1 :

pre: $C = (\Sigma j :: x_j) \;\wedge\; y = 0 \;\wedge\; (\forall j :: b_j)$

Q0: $C = y + (\Sigma j : b_j : x_j)$

Q1: $(\forall i, j : i{\leftarrow}j : b_i \Leftarrow b_j)$

post: $(\forall j :: \neg b_j)$

$add{\cdot}R$: $\{\ b_R\ \}\{\bullet\ (\forall j : R \leftarrow j : \neg b_j)\ \bullet\}$
$y, b_R := y + x_R$, false
$\{\ \neg b_R\ ,$ hence: $C = y\ \}$

and for each q different from R:

$add{\cdot}q$: $\{\ b_q\ \}\{\bullet\ (\forall j : q \leftarrow j : \neg b_j)\ \bullet\}$
$p :\ p \leftarrow q$
; $\{\ b_q\ \}\{\ p \leftarrow q$, hence: $p \neq q \wedge b_p\ \}\{\ (\forall j : q \leftarrow j : \neg b_j)\ \}$
$x_p, b_q := x_p + x_q$, false
$\{\ \neg b_q\ \}$
□

1.3 progress

In this simple case, progress can be formulated as a safety property. Informally, the shape of the proof obligation for progress is:

"the final state has not been reached" $\Rightarrow$

"the guard of at least one operation is true"

Because, for every q, component $add{\cdot}q$ has $(\forall j : q \leftarrow j : \neg b_j)$ as its guard, the proof obligation to demonstrate progress for our second approximation is:

$$(\exists i :: b_i) \quad \Rightarrow \quad (\exists i ::\ b_i\ \wedge\ (\forall j : i \leftarrow j : \neg b_j)\ ) \quad .$$

By means of contraposition and predicate calculus, this can be rewritten as:

$$(\forall i :: \neg b_i) \quad \Leftarrow \quad (\forall i ::\ \neg b_i \Leftarrow (\forall j : i \leftarrow j : \neg b_j)\ ) \quad .$$

Now, this is just the proposition that the relation $\leftarrow$ admits proofs by Mathematical Induction – read $i \leftarrow j$ as "j less than i", if you like –, which is equivalent to the proposition that the relation $\leftarrow$ is Well-Founded.

Because the domain of $\leftarrow$ – the set of nodes – is finite, Well-Foundedness of $\leftarrow$ is equivalent to the requirement that the relation $\leftarrow$ is *acyclic*, which means that $\overset{+}{\leftarrow}$ is irreflexive.

So, progress of our last approximation is guaranteed, provided $\leftarrow$ satisfies the additional requirement of *acyclicity*:

R1: $(\forall j :: \neg (j \overset{+}{\leftarrow} j))$.

Because, by the definition of transitive closure, $\leftarrow$ implies $\overset{+}{\leftarrow}$, we have that acyclicity of $\leftarrow$ implies irreflexivity of $\leftarrow$, as required.

1.4 epilogue

The correctness of the program for the summation phase depends on the following properties of the relation $\leftarrow$.

R0: $(\forall j : R \neq j : R \overset{+}{\leftarrow} j)$ *connectivity*

R1: $(\forall j :: \neg (j \overset{+}{\leftarrow} j))$ *acyclicity*

We may view the set of nodes together with $\leftarrow$ as a *directed graph*. In the jargon of graphs property R0 states that the root node R is *reachable* from every other node in the graph, whereas R1 states that this graph is *acyclic*.

* * *

The implementation of the shared variables in the algorithm requires communications over the network on which the algorithm will be implemented. In the current algorithm, every two components sharing variables are related by $\leftarrow$. For the sake of implementability we now require $\leftarrow$ to be such that related pairs of nodes are connected by a communication link in the network. This is formally rendered by requirement R2, in which "being connected by a communication link" is denoted by $\sim$.

R2: $(\forall i, j : i \leftarrow j : i \sim j)$ *compliance*

The *smallest* possible graph having these properties is a rooted, directed, spanning tree over the network. In all other presentations of this algorithm I have seen [2, 3, 7], this tree is introduced right from the start, and its necessity is taken for granted. Our development shows that, at least up to this point in the development, there is no need to restrict this graph to such a tree.

Of course, because in $add{\cdot}q$ exactly one successor p of q is chosen to which x_q is communicated, the actual communication pattern along which the numbers are collected is a tree indeed. The above development shows, however, that the choice of p in $add{\cdot}q$ can be postponed to the very last moment: there is no need to fix the tree in an earlier stage of the computation.

In addition, we now obtain a variation of the current solution for free: x_q may be split into several numbers, their sum equal to x_q, and these individual numbers may be added one by one to x_p; in each such addition a *different* successor p may be chosen! This may even be of some practical value, for instance, when the values involved are not numbers but large sets of data: such splitting up may then contribute to a better load balancing in the underlying network.

remarks: Whether or not such a variation is of practical value is irrelevant here: in many practical implementations the a priori restriction to a tree may be perfectly defensible. What is important, though, is that design decisions are made explicit as much as possible, and the above shows that the restriction to a tree is a true design decision.

When I was making preparations to write this report, I still believed that in the second part – the connection phase – I would inevitably run into the need to restrict as yet the connecting graph to a tree. To my own surprise, this did not happen!

□

Chapter 2

The Connection Phase

2.0 specification

The algorithm for the summation phase requires the existence of a binary relation $\leftarrow$ on the nodes, satisfying the following three requirements:

R0:	$(\forall j : R \neq j : R \overset{+}{\leftarrow} j\ )$	*connectivity*
R1:	$(\forall j :: \neg\, (j \overset{+}{\leftarrow} j)\ )$	*acyclicity*
R2:	$(\forall i, j : i \leftarrow j : i \sim j\ )$	*compliance*

The purpose of the connection phase is to compute such a relation; in this phase $\leftarrow$ is a variable, of type boolean function on pairs of nodes, and to be given such a value that the three requirements are met. We assume that initially no pairs of nodes are related by $\leftarrow$ at all. So, the specification of the connection phase is:

pre:	$(\forall i, j :: \neg\, (i \leftarrow j)\ )$
post:	R0 $\wedge$ R1 $\wedge$ R2

To start with, we will develop a solution in isolation, without regard to how the connection phase will interact with the summation phase; this will be dealt with later, in Section 3.0. (Recall that in the development of the algorithm for the summation phase, we have treated $\leftarrow$ as a constant, which is not true anymore.)

As before, we will develop the solution in a step-by-step fashion, taking the three requirements into account one by one. Subsequently, we will prove progress of the solution.

2.1 connectivity

We begin with the requirement of connectivity:

R0: $(\forall j : R{\neq}j : R \overset{+}{\leftarrow} j)$.

This is a global requirement that can be met in either of two ways: by means of a system invariant, or by having it implied by local properties. In this case, the use of a system invariant is appriopriate, because the set of nodes j satisfying $R \overset{+}{\leftarrow} j$ can be built up incrementally. To represent this set we introduce a boolean variable c_j, one for every node j, and we choose as invariant:

Q2: $(\forall j : R{\neq}j \wedge \neg c_j : R \overset{+}{\leftarrow} j)$.

Now $(\forall j :: c_j) \Rightarrow$ Q2 and Q2 $\wedge$ $(\forall j :: \neg c_j) \Rightarrow$ R0 . So, the purpose of the algorithm is to set all booleans c to false under invariance of Q2.

Because R is excluded from the range of Q2, the assignment $c_R := \mathsf{false}$ leaves Q2 invariant without further precautions. For q, $R{\neq}q$, the precondition for $c_q := \mathsf{false}$ is $R \overset{+}{\leftarrow} q$, and to simplify this we derive:

$$
\begin{array}{ll}
 & R \overset{+}{\leftarrow} q \\
\equiv & \quad \{ \text{ transitive closure } \} \\
 & R \leftarrow q \ \vee \ (\exists i :: R \overset{+}{\leftarrow} i \ \wedge \ i \leftarrow q) \\
\equiv & \quad \{ \text{ range split } R{=}i \vee R{\neq}i \text{, and absorption } \} \\
 & R \leftarrow q \ \vee \ (\exists i : R{\neq}i : R \overset{+}{\leftarrow} i \ \wedge \ i \leftarrow q) \\
\Leftarrow & \quad \{ \text{ Q2 } \} \\
 & R \leftarrow q \ \vee \ (\exists i : R{\neq}i : \neg c_i \ \wedge \ i \leftarrow q) \\
\equiv & \quad \{ \text{ unabsorption and range unsplit, as above } \} \\
 & R \leftarrow q \ \vee \ (\exists i :: \neg c_i \ \wedge \ i \leftarrow q) \\
\Leftarrow & \quad \{ \text{ strengthening } \} \\
 & (\exists i :: \neg c_i \ \wedge \ i \leftarrow q) \quad .
\end{array}
$$

The condition thus obtained is established by any assignment $n \leftarrow q := \mathsf{true}$, for any n satisfying $\neg c_n$. In the following first approximation to a solution we use a construct for some $n : \neg c_n$ that is supposed to prescribe a nondeterministic selection of *one or more* values n satisfying $\neg c_n$; we do so because there is no logical necessity to restrict this selection to *exactly one* value n.

Of course, such selection is only possible if $(\exists i :: \neg c_i)$, whence the guard of *fire*$\cdot q$.

The algorithm is the parallel composition of components *fire*$\cdot q$, for all q. Notice that in any execution of the program, *fire*$\cdot R$ will take place first: thus, the whole computation is initiated by the root R.

summary 0 :

pre: $(\forall i, j :: \neg (i \leftarrow j))\ \wedge\ (\forall j :: c_j)$

Q2: $(\forall j : R \neq j \wedge \neg c_j : R \overset{+}{\leftarrow} j)$

post: $(\forall j :: \neg c_j)$, hence: R0

fire$\cdot R$:
$\{\ c_R\ \}$
$c_R := \mathsf{false}$
$\{\ \neg c_R\ \}$

and for each q different from R:

fire$\cdot q$:
$\{\ c_q\ \}\{\bullet\ (\exists i :: \neg c_i)\ \bullet\}$
for some $n :\ \neg c_n$
do $\{\ \neg c_n\ \}\ n \leftarrow q := \mathsf{true}$ od
; $\{\ c_q\ \}\{\ (\exists i :: \neg c_i \wedge i \leftarrow q)$, hence: $R \overset{+}{\leftarrow} q\ \}$
$c_q := \mathsf{false}$
$\{\ \neg c_q\ \}$

□

2.2 acyclicity

Now we turn to the requirement of acyclicity:

R1: $(\forall j :: \neg (j \overset{+}{\leftarrow} j))$.

Because of the negation in this formula, the use of an additional invariant to obtain R1 is less appropriate, in the current algorithm, at least. Therefore, we have to exploit our only other possibility, namely to reduce R1 to local properties:

$(\forall j :: \neg (j \overset{+}{\leftarrow} j))$
$\equiv$ { one-point rule, to create room for manipulation }

$$
\begin{array}{ll}
 & (\forall i,j : i{=}j : \neg\,(i \stackrel{+}{\leftarrow} j)\,) \\
\equiv & \quad \{\ \text{range-term trading}\ \} \\
 & (\forall i,j : i \stackrel{+}{\leftarrow} j : i{\neq}j\,) \\
\Leftarrow & \quad \{\ \text{introduction of integer function } f\text{, see below; Leibniz}\ \} \\
 & (\forall i,j : i \stackrel{+}{\leftarrow} j : f_i{\neq}f_j\,) \\
\Leftarrow & \quad \{\ \text{irreflexivity of } <\ \text{(on the integers)}\ \} \\
 & (\forall i,j : i \stackrel{+}{\leftarrow} j : f_i{<}f_j\,) \\
\Leftarrow & \quad \{\ \text{transitive closure: } <\ \text{is transitive}\ \} \\
 & (\forall i,j : i{\leftarrow}j : f_i{<}f_j\,) \quad .
\end{array}
$$

This calculation shows that we may conclude the requirement of acyclicity from the existence of an integer function f, satisfying:

Q3: $\quad (\forall i,j : i{\leftarrow}j : f_i{<}f_j\,) \quad .$

The simplest way to obtain such function is to have it constructed by the algorithm, together with relation $\leftarrow$; that is, we turn Q3 into an additional invariant.

The precondition of the algorithm implies Q3, and the additional precondition for $n{\leftarrow}q := \mathsf{true}$ is:

$$f_n < f_q \quad ,$$

which can be established by assignment of a "large enough" value to f_q. Assignments to f, however, are also restricted by Q3; the precondition for an assignment $f_q := E$ is:

$$(\forall j : q{\leftarrow}j : E < f_j\,) \quad \wedge \quad (\forall i : i{\leftarrow}q : f_i < E\,) \quad .$$

The second conjunct of this condition is easily satisfied by choosing E "large enough"; for the first conjunct, the simplest solution seems to be to make its range empty, that is, to see to it that no j satisfies $q{\leftarrow}j$. To this purpose we derive:

$$
\begin{array}{ll}
 & (\forall j :: \neg\,(q{\leftarrow}j)\,) \\
\Leftarrow & \quad \{\ \text{generalization, to eliminate } q\text{, using precondition } c_q\ \} \\
 & (\forall i,j : c_i : \neg\,(i{\leftarrow}j)\,) \\
\equiv & \quad \{\ \text{range-term trading}\ \} \\
 & (\forall i,j : i{\leftarrow}j : \neg c_i\,) \quad .
\end{array}
$$

This gives rise to the introduction of yet another invariant, Q4; this invariant brings about new proof obligations, but fortunately all loose ends meet now: initially, Q4 holds, and the only potentially harmful assignment is $n \leftarrow q := \mathsf{true}$, but this already has $\neg c_n$ as its precondition.

Q4: $\quad (\forall i, j : i \leftarrow j : \neg c_i)$.

Thus, we obtain a solution that guarantees connectivity and acyclicity. Notice that function f is an auxiliary variable that may be omitted from the final implementation. The initial values of f are irrelevant, but are assumed to be integer. The operation $\mathsf{increase}\ f_q$ is supposed to increase f_q to such an extent that $f_n < f_q$ is established.

summary 1 :

pre: $\quad (\forall i, j :: \neg (i \leftarrow j)) \ \wedge \ (\forall j :: c_j)$

Q2: $\quad (\forall j : R \neq j \wedge \neg c_j : R \overset{+}{\leftarrow} j)$

Q3: $\quad (\forall i, j : i \leftarrow j : f_i < f_j)$

Q4: $\quad (\forall i, j : i \leftarrow j : \neg c_i)$

post: $\quad (\forall j :: \neg c_j)$, hence: R0 $\wedge$ R1

fire$\cdot R$: $\quad \{\ c_R\ \}$
$c_R := \mathsf{false}$
$\{\ \neg c_R\ \}$

and for each q different from R:

fire$\cdot q$: $\quad \{\ c_q\ \}\{\bullet\ (\exists i :: \neg c_i)\ \bullet\}$
$\mathsf{for\ some}\ \ n :\ \neg c_n$
$\mathsf{do}\ \{\ c_q\ \}\{\ \neg c_n\ \}$
$\mathsf{increase}\ f_q$
$;\ \{\ c_q\ \}\{\ \neg c_n\ \}\{\ f_n < f_q\ \}$
$n \leftarrow q := \mathsf{true}$
od
$;\ \{\ c_q\ \}\{\ (\exists i :: \neg c_i \wedge i \leftarrow q)$, hence: $R \overset{+}{\leftarrow} q\ \}$
$c_q := \mathsf{false}$
$\{\ \neg c_q\ \}$

□

2.3 network compliance

The third, and final, requirement imposed on $\leftarrow$ is R2, network compliance. This is most easily implemented by turning it into a system invariant:

Q5: $\quad (\forall i,j : i \leftarrow j : i \sim j\,) \quad .$

Now $(\forall i,j :: \neg\,(i \leftarrow j)\,) \Rightarrow$ Q5, and for the assignment $n \leftarrow q := \mathsf{true}$ the additional precondition required by Q5 is $n \sim q$, which we establish, in *fire*$\cdot q$, by strengthening the guard and restricting the range of n with it. Of course, this influences progress, but that is the subject of the next subsection.

Thus, network compliance is implemented by restricting the freedom of choice for the predecessors of q to its neighbours, and this is about the only thing we can do.

summary 2 :

pre: $\quad (\forall i,j :: \neg\,(i \leftarrow j)\,) \;\wedge\; (\forall j :: c_j)$

Q2: $\quad (\forall j : R \neq j \wedge \neg c_j : R \overset{+}{\leftarrow} j\,)$

Q3: $\quad (\forall i,j : i \leftarrow j : f_i < f_j\,)$

Q4: $\quad (\forall i,j : i \leftarrow j : \neg c_i\,)$

Q5: $\quad (\forall i,j : i \leftarrow j : i \sim j\,)$

post: $\quad (\forall j :: \neg c_j)$, hence: R0 $\wedge$ R1 $\wedge$ R2

fire$\cdot R$:
$\{\ c_R\ \}$
$c_R := \mathsf{false}$
$\{\ \neg c_R\ \}$

and for each q different from R:

fire$\cdot q$:
$\{\ c_q\ \}\{\bullet\ (\exists i : i \sim q : \neg c_i)\ \bullet\}$
for some $n :\ n \sim q \wedge \neg c_n$
do $\{\ c_q\ \}\{\ n \sim q \wedge \neg c_n\ \}$
increase f_q
; $\{\ c_q\ \}\{\ n \sim q \wedge \neg c_n\ \}\{\ f_n < f_q\ \}$
$n \leftarrow q := \mathsf{true}$
od
; $\{\ c_q\ \}\{\ (\exists i :: \neg c_i \wedge i \leftarrow q)$, hence: $R \overset{+}{\leftarrow} q\ \}$
$c_q := \mathsf{false}$
$\{\ \neg c_q\ \}$

□

2.4 progress

If it were not for the requirement of network compliance, the progress discussion for this algorithm would be trivial, but now it is not.

The desired final state of the system is $(\forall j :: \neg c_j)$, so the proof obligation for progress is:

$$(\exists j :: c_j) \quad \Rightarrow \quad c_R \ \vee \ (\exists j :: c_j \wedge (\exists i : i \sim j : \neg c_i)) \quad ,$$

which is equivalent to:

$$(\exists j :: c_j) \ \wedge \ \neg c_R \quad \Rightarrow \quad (\exists j, i : i \sim j : c_j \wedge \neg c_i) \quad .$$

To demonstrate this we proceed as follows:

$$\begin{array}{ll}
 & (\exists j, i : i \sim j : c_j \wedge \neg c_i) \\
\equiv & \quad \{ \text{ mainly De Morgan (twice) } \} \\
 & \neg (\forall j, i : i \sim j : c_j \Rightarrow c_i) \\
\Leftarrow & \quad \{ \text{ transitive closure } \} \\
 & \neg (\forall j, i : i \stackrel{+}{\sim} j : c_j \Rightarrow c_i) \\
\equiv & \quad \{ \text{ the network is connected, see below } \} \\
 & \neg (\forall j, i :: c_j \Rightarrow c_i) \\
\Leftarrow & \quad \{ \text{ instantiation } i := R \ \} \\
 & \neg (\forall j :: c_j \Rightarrow c_R) \\
\equiv & \quad \{ \text{ De Morgan, as above, in reverse order } \} \\
 & (\exists j :: c_j \wedge \neg c_R) \\
\equiv & \quad \{ \ \wedge \text{ over } \exists \ \} \\
 & (\exists j :: c_j) \ \wedge \ \neg c_R \quad .
\end{array}$$

In a simple way, this derivation reveals formally what informally is obvious: to enable communication over the whole network, the network must be connected. The above shows that connectivity is needed for the sake of progress: if the network is not connected then some nodes will never make their *fire*-move. This is the *only* place in the whole development where we make an appeal to connectivity of the underlying network.

The use of the transitive closure makes it possible to define connectivity in a way that is more concise and technically more manageable than the traditional definition involving "paths" (which are awkward to formalize):

definition:

"the network is connected" $\equiv$ $(\forall i, j :: i \overset{+}{\sim} j)$

□

This is the only requirement imposed on the network. In particular, we do not need that the connection relation is irreflexive: nodes may be connected to themselves without any problem. This follows immediately from the assertions in component *fire*·q: the precondition of $n \leftarrow q$:= true is $c_q \wedge n \sim q \wedge \neg c_n$, which implies $q \neq n$.

Chapter 3

Integration and Implementation

3.0 a complete program

Up to this point, we have developed two programs, one for the connection phase and one for the summation phase. The interface between these phases is the relation $\leftarrow$, which is built up in the connection phase and used in the summation phase. In the development of the summation phase we have assumed $\leftarrow$ to be constant, which is not true anymore; therefore, we have to pay attention to how the two phases are to be combined into a single program. (Fortunately, $\leftarrow$ is the only variable common to the two phases.)

A correct but naive combination would be a strictly sequential one: the summation phase would then start after all operations of the connection phase would have terminated. This can be implemented by guarding the components of the summation phase with $(\forall j :: \neg c_j)$. This is awkward, though, because a distributed implementation of these guards – a kind of *termination detection* – is essentially the same as the problem we are solving. In addition, such a solution would be needlessly restrictive in its concurrency.

Much nicer is the program obtained by sequential composition of the individual components of the two phases. The complete program then is the parallel composition of components $sum{\cdot}q$, where $sum{\cdot}q$ is the sequential composition of $fire{\cdot}q$ – from Section 2.3 – and $add{\cdot}q$ – from Section 1.2 –, for every node q.

Because $\leftarrow$ is the only variable common to the two phases, and because $\leftarrow$ is not modified in the summation phase, the only invariants and assertions whose correctness must be reviewed, are the ones of the summation phase

containing $\leftarrow$. All other invariants and assertions remain correct. In the following approximation to the final program, these formulae have been labelled with **??**:

pre: $(\forall i,j :: \neg(i \leftarrow j)) \ \wedge \ (\forall j :: c_j) \ \wedge$
$C = (\Sigma j :: x_j) \ \wedge \ y = 0 \ \wedge \ (\forall j :: b_j)$

Q0: $C = y + (\Sigma j : b_j : x_j)$

Q1: $(\forall i,j : i \leftarrow j : b_i \Leftarrow b_j)$ **??**

Q2: $(\forall j : R \neq j \wedge \neg c_j : R \overset{+}{\leftarrow} j)$

Q3: $(\forall i,j : i \leftarrow j : f_i < f_j)$

Q4: $(\forall i,j : i \leftarrow j : \neg c_i)$

Q5: $(\forall i,j : i \leftarrow j : i \sim j)$

post: $(\forall j :: \neg b_j)$

$sum{\cdot}R$: $\{\ c_R\ \}\{\ b_R\ \}$
$c_R := \mathsf{false}$
; $\{\ \neg c_R\ \}\{\ b_R\ \}$
$\{\bullet\ (\forall j : R \leftarrow j : \neg b_j)$ **??** $\bullet\}$
$y, b_R := y + x_R, \mathsf{false}$
$\{\ \neg b_R$, hence: $C = y\ \}$

and for each q different from R:

$sum{\cdot}q$: $\{\ c_q\ \}\{\ b_q\ \}$
$\{\bullet\ (\exists i : i \sim q : \neg c_i)\ \bullet\}$
for some n: $n \sim q \wedge \neg c_n$
do $\{\ c_q\ \}\{\ b_q\ \}\{\ n \sim q \wedge \neg c_n\ \}$
increase f_q
; $\{\ c_q\ \}\{\ b_q\ \}\{\ n \sim q \wedge \neg c_n\ \}\{\ f_n < f_q\ \}$
$n \leftarrow q := \mathsf{true}$
od
; $\{\ c_q\ \}\{\ b_q\ \}\{\ (\exists i :: \neg c_i \wedge i \leftarrow q)$, hence: $R \overset{+}{\leftarrow} q\ \}$
$c_q := \mathsf{false}$
; $\{\ \neg c_q\ \}\{\ b_q\ \}$

$\{\bullet\ (\forall j : q \leftarrow j : \neg b_j)\ \mathbf{??}\ \bullet\}$
$p:\ p \leftarrow q$
; $\{\ b_q\ \}\{\ p \leftarrow q\ \mathbf{??}$, hence: $p \neq q \wedge b_p\ \}$
$\{\ (\forall j : q \leftarrow j : \neg b_j)\ \mathbf{??}\ \}$
$x_p, b_q\ :=\ x_p + x_q$, false
$\{\ \neg b_q\ \}$

* * *

We now investigate the formulae whose correctness must be reviewed.

re Q1 : The only interfering statement is $n \leftarrow q\ :=$ true , in $sum{\cdot}q$; the additional precondition for the invariance of Q1 is $b_n \Leftarrow b_q$, which in view of the already present assertion b_q is equivalent to b_n . Therefore, we add b_n as coassertion to $n \leftarrow q\ :=$ true , and for the sake of the validity of this b_n , we add $(\forall i : i \sim q : b_i)$ as preassertion to $sum{\cdot}q$; as a result, b_n is implied now (by this new preassertion and $n \sim q$), at the expense of a new proof obligation for $(\forall i : i \sim q : b_i)$ –see below– .

re $(\forall j : q \leftarrow j : \neg b_j)$, both for $R{=}q$ and for $R{\neq}q$: The only interfering statement is $q \leftarrow j\ :=$ true (that is, $n \leftarrow j\ :=$ true in component $sum{\cdot}j$, with q for n), which has c_j as a preassertion; hence, the rule of disjointness settles this, provided we add $(\forall j : q \leftarrow j : \neg c_j)$ as coassertion (and coguard) to all occurrences of $(\forall j : q \leftarrow j : \neg b_j)$. This new coassertion, however, also contains $\leftarrow$, which is awkward; to eliminate this $\leftarrow$ we strengthen –Q5!– the coassertion, by weakening its range from $q \leftarrow j$ to $q \sim j$ (thus replacing the variable $\leftarrow$ by the constant $\sim$). So, the coassertion (and coguard) becomes $(\forall j : q \sim j : \neg c_j)$; its advantage is that, as an assertion, it is trivially correct.

remark: This game is quite subtle: had we applied the strengthening trick directly to $(\forall j : q \leftarrow j : \neg b_j)$, yielding $(\forall j : q \sim j : \neg b_j)$, we would have obtained correct assertions, but the resulting guards would be too strong, giving rise to the danger of deadlock.

□

re $p \leftarrow q$: This is trivially correct because of "widening" : the assignments to $\leftarrow$ only truthify it.

re $(\forall i : i \sim q : b_i)$, as additional precondition of $sum{\cdot}q$: This is locally correct, as it is implied by the global precondition $(\forall j :: b_j)$. As for its global correctness, the only interfering statement is $b_i :=$ false , in component $sum{\cdot}i$. As a result of the previous addition, this assignment now has $(\forall j : i \sim j : \neg c_j)$

as a precondition, with $i{\sim}q \Rightarrow \neg\, c_q$ as an instance. Because $(\forall i : i{\sim}q : b_i)$ has c_q as a coassertion, the rule of disjointness does the job here too.

* * *

Thus, we arrive at our final version of the program.

a program for distributed summation:

pre: $(\forall i, j :: \neg\,(i{\leftarrow}j)\,) \;\wedge\; (\forall j :: c_j) \;\wedge$
$C \;=\; (\Sigma j :: x_j) \;\wedge\; y \;=\; 0 \;\wedge\; (\forall j :: b_j)$

Q0: $C \;=\; y \;+\; (\Sigma j : b_j : x_j)$

Q1: $(\forall i, j : i{\leftarrow}j : b_i \Leftarrow b_j\,)$

Q2: $(\forall j : R{\neq}j \wedge \neg c_j : R \overset{+}{\leftarrow} j\,)$

Q3: $(\forall i, j : i{\leftarrow}j : f_i < f_j\,)$

Q4: $(\forall i, j : i{\leftarrow}j : \neg\, c_i\,)$

Q5: $(\forall i, j : i{\leftarrow}j : i{\sim}j\,)$

post: $(\forall j :: \neg\, b_j)$

$sum{\cdot}R$:
$\{\; c_R \;\}\{\; b_R \;\}$
$c_R \;:=\; \mathsf{false}$
$;\; \{\; \neg\, c_R \;\}\{\; b_R \;\}$
$\{\bullet\; (\forall j : R{\sim}j : \neg\, c_j)\; \bullet\}\{\bullet\; (\forall j : R{\leftarrow}j : \neg\, b_j)\; \bullet\}$
$y, b_R \;:=\; y{+}x_R\,,\; \mathsf{false}$
$\{\; \neg\, b_R\;,\;$ hence: $C = y\;\}$

and for each q different from R:

$sum{\cdot}q:$ $\{\ c_q\ \}\{\ b_q\ \}\{\ (\forall i : i{\sim}q : b_i)\ \}$
$\{\bullet\ (\exists i : i{\sim}q : \neg c_i)\ \bullet\}$
for some $n:\ n{\sim}q \wedge \neg c_n$
do $\{\ c_q\ \}\{\ b_q\ \}\{\ n{\sim}q \wedge \neg c_n\ \}\{\ b_n\ \}$
increase f_q
$;\ \{\ c_q\ \}\{\ b_q\ \}\{\ n{\sim}q \wedge \neg c_n\ \}\{\ b_n\ \}\{\ f_n < f_q\ \}$
$n{\leftarrow}q := \mathsf{true}$
od
$;\ \{\ c_q\ \}\{\ b_q\ \}\{\ (\exists i :: \neg c_i \wedge i{\leftarrow}q)\ \}$, hence: $R^{+}{\leftarrow}q\ \}$
$c_q := \mathsf{false}$
$;\ \{\ \neg c_q\ \}\{\ b_q\ \}$
$\{\bullet\ (\forall j : q{\sim}j : \neg c_j)\ \bullet\}\{\bullet\ (\forall j : q{\leftarrow}j : \neg b_j)\ \bullet\}$
$p:\ p{\leftarrow}q$
$;\ \{\ b_q\ \}\{\ p{\leftarrow}q$, hence: $p{\neq}q \wedge b_p\ \}$
$\{\ (\forall j : q{\sim}j : \neg c_j)\ \}\{\ (\forall j : q{\leftarrow}j : \neg b_j)\ \}$
$x_p, b_q := x_p{+}x_q$, false
$\{\ \neg b_q\ \}$

□

The guard $\{\bullet\ (\forall j : q{\sim}j : \neg c_j)\ \bullet\}$ and its coguard $\{\bullet\ (\forall j : q{\leftarrow}j : \neg b_j)\ \bullet\}$ can be combined into a single one, thus:

$$\{\bullet\ (\forall j : q{\sim}j : \neg c_j \wedge (\neg(q{\leftarrow}j) \vee \neg b_j)\ )\ \bullet\} \quad .$$

3.1 shared variables and communication

Here we discuss, rather informally, how the shared variables, and the guards containing them, in our final program can be implemented by means of communication over the links between the nodes of the network. A formal treatment is straightforward, as this poses no additional difficulties.

We consider *asynchronous* communication between a pair of (connected) nodes: the one node can *send* a *message* that, after some finite delay, can be *received* by the other node. Here "asynchronous" means that the sending node is not synchronised with the receiving node: after sending the message the sender may proceed immediately. The receiving node, however, does need synchronisation, for messages cannot be received before they have been sent.

The main principle for this kind of communication is the "Rule of Import and Export" : a postassertion B of a receive operation is locally correct provided the corresponding send operation has B as a correct preassertion. (In

addition, B must also be globally correct, but this is not what the Rule of Import and Export is about.) Thus, the validity of an assertion that has been established in one node can be communicated to another node by means of a (possibly empty) message. Moreover, the content of that message may carry additional data, like the value of a private variable of the sender.

It is important to keep in mind that, formally, send and receive operations play different – complementary – roles; receive operations are needed for partial correctness whereas send operations are needed for progress: a correct program from which all send operations are removed remains partially correct but will suffer from deadlock.

* * *

The shared variables in the program are the booleans b, c and $\leftarrow$, and the integers x. For their distributed implementation, we discuss them one by one, and we do so "from the point of view of" component $sum{\cdot}q$. First, we will introduce all necessary receive operations; second, only then will we introduce the required send operations. This transformation is almost completely forced, in that, apart from minor variations, there is hardly anything else we can do; the variation chosen here uses a minimal number of messages.

re c : In component $sum{\cdot}q$, the only assignment to c is $c_q := \mathsf{false}$, and the only inspections of c occur in the guards $\{\bullet\ (\exists i : i{\sim}q : \neg\, c_i)\ \bullet\}$ and $\{\bullet\ (\forall j : q{\sim}j : \neg\, c_j)\ \bullet\}$. So, $sum{\cdot}q$ needs c_j only from neighbours j. Also, $\neg\, c_j$ is established by $sum{\cdot}j$, in its assignment $c_j := \mathsf{false}$.

To detect $\neg\, c_j$ component $sum{\cdot}q$ receives one message from $sum{\cdot}j$, one for every neighbour j. So, in total $sum{\cdot}q$ receives as many messages as it has neighbours. With N_q for the (constant) number of neighbours of node q, the guard $\{\bullet\ (\exists i : i{\sim}q : \neg\, c_i)\ \bullet\}$ can be implemented as
$\{\bullet$ "at least one message has been received" $\bullet\}$, and the guard
$\{\bullet\ (\forall j : q{\sim}j : \neg\, c_j)\ \bullet\}$ can be implemented as
$\{\bullet$ "N_q messages have been received" $\bullet\}$.

re b : In the same vein, the only assignment (in $sum{\cdot}q$) is $b_q := \mathsf{false}$, and the only values needed in $sum{\cdot}\, q$ are b_j for neighbours j, namely in the guard $\{\bullet\ (\forall j : q{\sim}j : \neg(q{\leftarrow}j) \vee \neg\, b_j)\ \bullet\}$.

We could introduce new messages to signal that $\neg\, b_j$ has been established, but it is simpler to let these messages signal that the whole disjunction $\neg(q{\leftarrow}j) \vee \neg\, b_j$ has been established. Because this occurs in conjunction with $\neg\, c_j$, we combine the corresponding messages. So, we decide that $sum{\cdot}q$ still receives a single message per neighbour, and from neighbour j

this message now conveys $\neg c_j \wedge (\neg(q \leftarrow j) \vee \neg b_j)$; as a result, the new guard $\{\bullet$ "N_q messages have been received" $\bullet\}$ implies this for all neighbours j .

re $\leftarrow$: In component $sum{\cdot}q$, the only assignment to $\leftarrow$ is $n \leftarrow q := \mathsf{true}$, and the only inspection of $\leftarrow$ occurs in the selection $p{:}\ p \leftarrow q$. (Of course, $\leftarrow$ also occurs in the guards $\neg(q \leftarrow j) \vee \neg b_j$ but this has been taken care of already.) So, we can represent $\leftarrow$ in a distributed way, by having component $sum{\cdot}q$ keep the subset of neighbours i satisfying $i \leftarrow q$ –that is, its successorset– in a private variable; thus, shared variable $\leftarrow$ is represented by a collection of private variables, one per node.

re send operations : Every component receives a message from each of its neighbours, so every component must also send a message to each of its neighbours. According to the Rule of Import and Export, sending a message from node q to neighbour i must have $\neg c_q \wedge (\neg(i \leftarrow q) \vee \neg b_q)$ as a precondition. The conjunct $\neg c_q$ implies that this operation may only take place *after* the assignment $c_q := \mathsf{false}$. The conjunct $\neg(i \leftarrow q) \vee \neg b_q$, together with the general desire (for the sake of progress) to have the message sent as early as possible, gives rise to case analysis. After the assignment $c_q := \mathsf{false}$ the set of successors of q is stable, and for every neighbour i satisfying $\neg(i \leftarrow q)$ the message can be sent immediately, whereas for neighbour i with $i \leftarrow q$ the message can only be sent after completion of $b_q := \mathsf{false}$.

re x : We distribute the variables x over the components, in such a way that x_q is a private variable of $sum{\cdot}q$. So, x_p is a private variable of $sum{\cdot}p$ now, and the assignment $x_p := x_p + x_q$ must be implemented in $sum{\cdot}p$, as an assignment $x_p := x_p + z_{p,q}$, where $z_{p,q} = x_q$. Variable $z_{p,q}$ models communication from q to p: as the addition now is performed in $sum{\cdot}p$, the value of x_q must be sent to node p . Because $p \leftarrow q$, this value can be sent in the message that is sent to p anyhow, right after $b_q := \mathsf{false}$.

exercise: We have tacitly used that the neighbour relation $\sim$ is *symmetric*: where and how?
□

3.2 summary

In the following program, all assertions and all auxiliary variables have been removed. A "message" either is a "signal" or a "response"; a "signal" carries no data, whereas a "response" sent by node q carries the value of x_q . To keep the formulae simple, we have assumed $z_{q,j} = 0$ whenever the message to

q from j is a "signal" (and, otherwise, $z_{q,j} = x_j$, as required). (Recall that constant N_q is the number of neighbours of q.)

$sum{\cdot}R$: for all $n : n \sim R$ do "send signal to n" od
; {• "N_R messages have been received" •}
; $x_R := x_R + (\Sigma j : R \sim j : z_{R,j})$
; $y := y + x_R$
{ $C = y$ }

and for each q different from R:

$sum{\cdot}q$: {• "at least one message has been received" •}
; for some n: "a message has been received from n"
do $n \leftarrow q := \mathsf{true}$ od
; for all $n : n \sim q \wedge \neg(n \leftarrow q)$ do "send signal to n" od
; {• "N_q messages have been received" •}
; p: $p \leftarrow q$
; $x_q := x_q + (\Sigma j : q \sim j : z_{q,j})$
; $z_{p,q} := x_q$; "send response to p"
; for all $n : n \leftarrow q \wedge n \neq p$ do "send signal to n" od

exercise: In view of the desire to have messages sent as early as possible, one might consider the following program, obtained from the above one by reshuffling statements that do not seem to influence one another; as a result, the $N_q - 1$ "signals" sent by q are sent as early as possible, for $R \neq q$:

$sum{\cdot}q$: {• "at least one message has been received" •}
; for some n: "a message has been received from n"
do $n \leftarrow q := \mathsf{true}$ od
; p: $p \leftarrow q$
; for all $n : n \sim q \wedge n \neq p$ do "send signal to n" od
; {• "N_q messages have been received" •}
; $x_q := x_q + (\Sigma j : q \sim j : z_{q,j})$
; $z_{p,q} := x_q$; "send response to p"

Investigate whether, and if possible prove that, this version is correct.
□

Chapter 4

Discussion

A formal step-by-step development of the algorithm for distibuted summation is certainly feasible and reveals much more of the structure of the design than an a-posteriori proof ever can. Even during the act of writing this report I occasionally changed my mind, as my own understanding of the design evolved during the process.

This study distinguishes itself from other presentations [2, 3, 7] of the same algorithm mainly by its length. A formal step-by-step development requires more space than just an a-posterori proof of correctness, because of the heuristic considerations involved and because of the summaries concluding the steps. In addition, the presentations in [2, 3, 7] do not contain the proofs of all relevant properties, whereas I have pursued completeness to a greater extent.

The other presentations define correctness in terms of the execution traces of the program. This is rather strange, first, because it is unclear what the execution traces of a (nonexecutable!) specification should be and, second and more important, because the relation between the programming formalism and its execution traces is independent of any particular program and can be fixed once and for all: the proof rules of the formalism thus make up the interface between the use of the formalism and its implementation.

It is somewhat amazing to observe how easily other authors take aspects for granted that are so obviously irrelevant. For example, although every communication link carries exactly one message (in either direction) and, hence, no reason exists to discuss the order of delivery of multiple messages, at least two authors [3, 7] assume the links to have first-in-first-out delivery.

Also, some authors [2, 3] assume that no node in the network has a link to itself. The author of [2] needs this to avoid deadlock, due to his use of *synchronous* communication, but this only shows that synchronous communi-

cation synchronises the processes more strictly than is necessary or desirable.

In an Owicki-Gries style of reasoning asynchronous communication poses no formal problems, particularly so in combination with stable predicates. It is my definite opinion that asynchronous communication is more fundamental and simpler than synchronous communication; for example, synchronous communication can be implemented easily by means of a "hand-shake" composed from two asynchronous communications, whereas the converse implementation is impossible without the introduction of additional processes. Moreover, for distributed systems with nonnegligible communication delays, asynchronous communication is more realistic.

In the same vein, quite unnecessary design decisions are taken. For example, sometimes [2, 3, 7] the sender of the very *first* message received is designated as the receiver's father (in the spanning tree); particularly in asynchronous systems this firstness of arrival has no intrinsic meaning as it carries no information about (the state of) the sender of the message.

Two authors [3, 7] report to have problems with nondeterminism; to me it remains unclear why, but insufficient abstraction from the computational model – the execution traces – may have something to do with this. In the Owicki-Gries formalism nondeterminism poses no problems, because the formalism has been designed to cope with it. The proof obligations of a program according to this formalism are just a collection of propositions about atomic statements, which can be verified, mechanically or manually, by the normal rules of logic.

Because the state of (progress of) a computation comprises not only the values of the variables but also the "control state" of that process, we need some notational device to incorparate, wholly or partly, information about this control state in the assertions. That this is unavoidable can even be proved [4, 1]. For this purpose, auxiliary variables can be used, or dedicated (so-called) *control predicates* [5]. Auxiliary variables are attractive because they can be introduced (and not introduced) exactly to the extent in which they are needed (and not needed). In our development of distributed summation, for example, the booleans b and c are auxiliaries representing the control state of the nodes. More important than one's preference for one notational device over the other, however, is the recognition that sometimes assertions must refer to the control state as well.

In the algorithm for distributed summation every communication link carries only one message in either direction, and this makes a formal discussion of communication simple. A viable formal technique to cope with several messages along the same link may be to provide the messages with unique names,

such that the sending and receiving of every individual message can be discussed in isolation. To start with, such identification plays a role similar to auxiliary variables, namely to simplify reasoning; whether or not these names must be really implemented will depend on how the algorithm is developed. This technique is a subject of current investigations.

Bibliography

[1] W.H.J. Feijen, A.J.M. van Gasteren, *On a Method of Multiprogramming*. Springer-Verlag, New York, 1999.

[2] J.F. Groote, F. Monin, J. Springintveld, *A Computer Checked Algebraic Verification of a Distributed Summation Algorithm*.
Computing Science Report 97-14, Eindhoven University of Technology, 1997.

[3] W.H. Hesselink, *A Mechanical Proof of Segall's PIF Algorithm*.
Formal Aspects of Computing **9**(2), pp. 208-226, 1997.

[4] R.R. Hoogerwoord, *Sometimes auxiliary variables are necessary*.
memorandum rh185, Eindhoven University of Technology, 1993.

[5] L. Lamport, *Control Predicates are Better than Dummy Variables for Reasoning about Program Control*.
ACM Transactions on Programming Languages and Systems **10**(2), pp. 267-281, 1988.

[6] S. Owicki, D. Gries, *An axiomatic proof technique for parallel programs I*. Acta Informatica **6**, pp. 319-340, 1976.

[7] F.W. Vaandrager, *Verification of a Distributed Summation Algorithm*, in: I. Lee, S.A. Smolka (editors), *CONCUR'95: Concurrency Theory*, LNCS 962, Springer-Verlag, Berlin etc., 1995.