Derivation of Distributed Programs in the Theory of Owicki and Gries: An Example

Doug Goldson
School of ITEE,
University of Queensland,
Queensland 4072,
Australia.
goldson@itee.uq.edu.au

Abstract

This paper describes the derivation of a program for the propagation of information over a network, with acknowledgement (feedback) when the computation is complete. The derivation is carried out in the theory of Owicki and Gries. The paper therefore illustrates the use of this theory for the *derivation*, as opposed merely to the verification, of *distributed* multiprograms. Notable is that the derivation, while calculational in style, is carried out with a minimum of formal machinery, e.g., there is no temporal logic. The derivation also serves as a concrete illustration of program reuse. A theory that is based on a shared variable model of communication is shown to manage the design of distributed multiprograms quite well.

Keywords: program derivation, distributed programs, Owicki and Gries

1. INTRODUCTION

This paper applies the theory of Owicki and Gries [6, 2, 3] to a problem of distributed multiprogramming. The purpose is to show how the theory can be used as a method of *distributed* multiprogram derivation. To this extent it is a companion to [4] which was written to highlight the usefulness of the theory for deriving multiprograms as opposed to merely verifying them. The theory is simple – proofs are carried out within the programming language using the assertional Hoare style – and old – it was developed in the 1970s. However, recent developments by Feijen and van Gasteren [3] have shown us how to derive multiprograms, rather than merely to verify them. What is more, the design heuristics that are used in these derivations are attractively simple. This exercise is motivated by [5] where Petri nets are used as a means to verify distributed programs. Important differences with [5] are that: here a program is derived and not merely verified; reasoning is applied to program statements, rather than to some other program model such as a Petri net; and temporal logic is avoided. Noteworthy too is that the exercise shows how distributed programs can be derived within a programming model that is based on communication using shared variables.

Section 2 describes the problem: to design a program for the propagation of information over a network, with acknowledgement when the computation is complete. Sections 3 and 4 describe the derivation and Sections 3.1 and 4.1 draw conclusions from the exercise. Finally, Section 4.3 defines the atomicity assumptions of the programming model in a way that confirms the foregoing derivation. The rest of this section briefly describes the programming model and the theory of Owicki and Gries. This is meant to be sufficient to follow the derivation, but see [3] for an impeccable treatment of the theory of Owicki and Gries.

A multiprogram consists of more than one component program (henceforth, called a component) to be executed at the same time. Instead of saying that more than one component is ‘executing at the same time’, it is equivalent (and simpler) to say that at most one component is executing an

atomic action, but the choice of executing component is not determined by the multiprogram text (see, for example, [7](p3) and [3](p3)). However, the choice is assumed to be *fair*, in the sense that no component that can execute an action is forever prevented from doing so. The programming language is Dijkstra's guarded commands [1], which supports condition synchronisation with the blocking conditional

if $B \rightarrow \text{skip}$ **fi**, where program control reaches *skip* if B is evaluated when true and it remains at the guard if B is evaluated when false. The instruction must itself be *fair*, in the sense that a guard can not remain unevaluated forever.

The theory of Owicki and Gries is based on the idea of a correct multiprogram annotation (in the style of Hoare logic). An annotation is correct when it is both *locally* correct and *globally* correct. For action S in component p annotated as a Hoare triple $\{P\} S \{Q\}$, postcondition Q is locally correct when precondition P establishes the weakest liberal precondition of S and Q [1], i.e., when

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

Furthermore, assertion Q is globally correct when no action A with precondition R in another component q is capable of falsifying Q , i.e., when

$$R \wedge Q \Rightarrow wlp.A.Q$$

and similarly for P : P in component p is globally correct when

$$R \wedge P \Rightarrow wlp.A.P$$

In the derivation to come, familiarity with Hoare logic will be assumed and the proof of local correctness will largely be left to the reader. This will allow greater attention to be paid to proof of global correctness, the novel aspect of the theory.

A distinction important to the derivation to come is that of shared and private variables. A *private* variable is a variable written by just one component, and so private to that component. A *shared* variable is a variable written by more than one component. Private variables, as we shall see, have the attractive property of the [3](p43)

Rule of Private Variables: An assertion in a component that only depends on private variables of that component is globally correct.

Since no other component can change the variables, no other component can falsify the assertion that depends upon them.

2. SPECIFICATION OF A MULTIPROGRAM TO PROPAGATE INFORMATION WITH FEEDBACK (PIF)

The program will be derived in an abstract way and it will later be shown (Section 3.1) how to interpret it as propagating information with feedback. The program is distributed over a fixed, connected network of $N > 0$ nodes, of which one node is distinguished as the *starter*, all others being *followers*. The requirements are

- (0) Each node can only communicate with its neighbours.
- (1) The starter starts before any follower starts.
- (2) The starter terminates after all followers terminate.
- (3) The starter terminates.

Requirements (0) and (1) are requirements of program topology. (0) may even seem too obvious to write down. However, in the theory of Owicki and Gries, components communicate using program variables and we do well to include an explicit check that these variables can only be shared between network neighbours. Requirement (1) is also a requirement on the shape of a solution: the multiprogram starts at *one* starter node. All the work of the derivation lies in requirements (2)

and (3). (2) is the safety requirement that the starter can not terminate until all other components terminate. It is a safety property because it does not mean that all other components do terminate. Requirement (3) is the progress requirement that the starter does terminate, from which it follows, by (2), that the multiprogram as a whole terminates.

3. DERIVATION 1

Let relation n define the network of N nodes, such that

$$n.p.q = \text{node } p \text{ is connected to node } q$$

n is symmetric and antireflexive.

The derivation begins with a program ST to compute a network spanning tree. The program is taken from [3](p303). Component R is the code of the root node. The other $N - 1$ components are symmetric and are represented by component p . Note that each component q has a shared variable $v.q$, which is used to communicate with its neighbours, and a private variable $f.q$ which records its father in the spanning tree. The types of $v.q$ and $f.q$ are ‘node identifier’ plus $\perp$, a constant distinct from the node identifiers. For example, since the node identifiers range over the set $0 \leq q < N$, $\perp$ can take the value N .

Note that a quantified assertion such as $\forall q : P : Q$ is read ‘Every q that satisfies P , satisfies Q ’, so that

$$\forall q : P : Q \quad = \quad \forall q :: P \Rightarrow Q$$

Further, since all quantified variables q range over the set $0 \leq q < N$, we will abbreviate $\forall q : 0 \leq q < N : Q$ to $\forall q :: Q$. Finally, the program itself is sugared so that, for example, code ‘do $v.q := p$ for all neighbours q of p ’ is abbreviated to $(\forall q : n.q.p : v.q := p)$.

ST

Pre: $\forall q :: v.q = \perp$

Comp. R : $(\forall q : n.q.R : v.q := R)$

Comp. p : **if** $v.p \neq \perp \rightarrow \text{skip}$ **fi**
 $(p \neq R) \quad f.p := v.p$
 $(\forall q : n.q.p : v.q := p)$

ST satisfies

- (4) If ST terminates variables $f.q$ define a spanning tree.
- (5) ST terminates.

Note that ST already satisfies three requirements of the PIF program. (0) is met because all communication is through array v and assignment by p to v is restricted to the neighbours of p . (1) is met if Comp. R is the starter. And (3) is met because of (5). This leaves only (2). Anticipating later needs, and by way of illustrating the idea of a correct annotation, Comp. p is annotated

Pre: $\forall q :: v.q = \perp \wedge f.q = \perp$

Comp. R : $(\forall q : n.q.R : v.q := R)$

Comp. p : **if** $v.p \neq \perp \rightarrow \text{skip}$ **fi** $\{v.p \neq \perp\}$
 $(p \neq R) \quad \{f.p = \perp\} f.p := v.p \{f.p \neq \perp\}$
 $(\forall q : n.q.p : v.q := p)$

The annotation is made locally correct by strengthening the ST precondition with $f.q = \perp$. $v.p \neq \perp$ is globally correct by the rule of widening ($v.p$ is never assigned $\perp$), and $f.p = \perp$ and $f.p \neq \perp$ are globally correct by the Rule of Private Variables. Note that no changes have been made to the ST code, and no changes will be made to it, so the code can now be abbreviated

Comp. R : RST

Comp. p : pST
($p \neq R$)

Requirement (2) is now met by introducing a private variable t to each component, under invariant T

Inv: $T: \forall q :: q \neq R \Rightarrow (t.q = \text{Comp}.q \text{ has terminated})$

The annotation now shows Comp. R to wait for all other nodes to terminate

Pre: $\forall q :: v.q = \perp \wedge f.q = \perp \wedge \neg t.q$

Comp. R : $RST \text{ if } \forall q : q \neq R : t.q \rightarrow \text{skip} \text{ fi } \{\forall q : q \neq R : t.q\}$

Comp. p : $pST \ t.p := \text{true} \ \{t.p\}$
($p \neq R$)

Local correctness of T is proved by the location of $t.p := \text{true}$. Global correctness is automatic because T is invariant. Global correctness of the assertion in Comp. R is by the rule of widening ($t.q$ is never falsified). Or, in other terms, by T , because a terminated component can not be unterminated!

This change to the code introduces a common situation in multiprogramming whereby one requirement is met at the cost of another. Topology Requirement (0) is now true only if every non-root node is a neighbour of Comp. R . However, given that the ST code computes a spanning tree,

(2): Comp. R terminates after all other nodes
 $\Leftarrow \{(4,5): \text{Comp}.R \text{ is the root of a spanning tree}\}$
 Each node terminates after its children

, it is enough to arrange each node to terminate after its children (a set of its neighbours). This leads to investigating the conditions for assertion ' q is a child of p that has terminated' to be computed. It is true when q has a father and if the father is p then q has terminated

$CT.p : \forall q :: CT.q.p$

$CT.q.p : (n.q.p \wedge q \neq R) \Rightarrow (f.q \neq \perp \wedge (f.q = p \Rightarrow t.q))$

This gives a new annotation for Comp. p

Comp. p : pST
($p \neq R$) $\text{ if } CT.p \rightarrow \text{skip} \text{ fi}$
 $\{CT.p\}$
 $t.p := \text{true}$

and $CT.p$ is globally correct when $CT.q.p$ is globally correct for all q , which follows from the following instance of the global correctness rule (refer to Section 1), where the action is assignment $f.q := v.q$ in Comp. q , and its precondition is $f.q = \perp$

$$\begin{aligned}
 f.q &= \perp \wedge CT.q.p \\
 &= \\
 f.q &= \perp \wedge (n.q.p \wedge q \neq R \Rightarrow f.q \neq \perp \wedge (f.q = p \Rightarrow t.q)) \\
 &\Rightarrow \\
 \neg(n.q.p \wedge q \neq R) \\
 &\Rightarrow \\
 n.q.p \wedge q \neq R &\Rightarrow v.q \neq \perp \wedge (v.q = p \Rightarrow t.q) \\
 &\Rightarrow \\
 wlp.(f.q := v.q).(CT.q.p)
 \end{aligned}$$

In words, if a father is assigned to q , it is not p because q is not in the neighbourhood of p .

Requirements (2) and (0) are now both met and the annotation shows a component to wait for its children to terminate

1 (Program)

Inv: $T: \forall q :: q \neq R \Rightarrow t.q = \text{Comp}.q$ has terminated
 Pre: $\forall q :: v.q = \perp \wedge f.q = \perp \wedge \neg t.q$

Comp. R : RST
 if $CT.R \rightarrow skip$ fi $\{CT.R\}$

Comp. p : $pST \{f.p \neq \perp\}$
 $(p \neq R)$ if $CT.p \rightarrow skip$ fi $\{CT.p\}$
 $t.p := true$

Because additional synchronisation code has been introduced into all components, it remains to check Requirement (3), that $\text{Comp}.R$ still terminates. This is done by contradiction

Comp. R does not terminate
 = $\{(5): RST \text{ terminates}\}$
 Comp. R is blocked at $CT.R$ guard
 = $\{\text{Comp}.p \text{ annotation: } \forall q : q \neq R : f.q \neq \perp \text{ is sometime stably true}\}$
 Some child q of R does not terminate ($f.q = R \wedge \neg t.q$ is stably true)
 = $\{(5): pST \text{ terminates}\}$
 Some child r of q does not terminate
 = $\{\text{Same argument}\}$
 $\dots$
 = $\{(4,5): f \text{ define a finite tree}\}$
 $false$

In words, the assumption that $\text{Comp}.R$ does not terminate allows one to construct an infinite branch in a tree. Since the network is finite, $\text{Comp}.R$ must terminate.

3.1. Conclusions from derivation 1

The goal was to derive a PIF program, in which a starter node propagates a message M over a network and then waits for acknowledgement that all other nodes have received it. $\text{Comp}.R$ is the starter with M . Associate the action ' p sends M to neighbour q ' with $v.q := p$. All nodes receive M because $\text{Comp}.p$ does not send a message before receiving M . Associate the action 'acknowledge M ' with $t.p := true$. $\text{Comp}.p$ does not acknowledge M before acknowledgement from its children. When $\text{Comp}.R$, at the root of the tree, receives acknowledgement from its children, all other nodes have acknowledged M .

Choosing ST as the starting point of the derivation is not cheating. Rather, it is an example of what software engineers call 'program reuse' and what mathematicians call 'using a lemma'.

Given that a PIF program must reach all nodes in a network, the computation of a spanning tree is a natural starting point. Indeed, the PIF program reflects a very effective separation of concerns: in a program to (a) propagate information over a network and (b) wait for an acknowledgement, it makes good sense to tackle these two aspects in isolation from each other. Aspect (a) was solved at once with an ‘off the shelf’ solution courtesy of [3], and the strategy has paid off, particularly in proving the progress Requirement (3), which is the area where the theory of Owicki and Gries is at its weakest. Given that Program *ST* terminates, then individual deadlock can only arise at the *CT.q* guard.

4. DERIVATION 2

The PIF program verified in [5] is the echo program (see also ([8], p194)). The next exercise is to derive this program. This is an exercise of a different sort, because now there is a target solution in mind, with topology: each node terminates after a message from all its neighbours. Therefore, we aim to show

Comp.*R* terminates after all other nodes
 $\Leftarrow$ {Spanning tree}
 Each node terminates after its children
 $\Leftarrow$ {(*) : To be arranged}
 Each node terminates after a message from all its neighbours

This will be done by coordinate transformation on Program (1). First, the shared variable *v.p* of Comp.*p* will be replaced by a set of new private (to *q*) variables *s.q.p*, where *s.q.p* can be interpreted as ‘*q* sends a message to *p*’ and, equivalently, ‘*p* receives a message from *q*’. The replacement is done according to the coupling invariant *Q*

Inv: $T: \forall q :: q \neq R \Rightarrow t.q = \text{Comp}.q \text{ has terminated}$
 $Q: \forall p :: v.p \neq \perp = \exists q :: s.q.p$
 Pre: $\forall q :: v.q = \perp \wedge f.q = \perp \wedge \neg t.q \wedge \forall p :: \neg s.p.q$

Comp.*R*: $(\forall q : n.q.R : s.R.q, v.q := \text{true}, R)$
 if *CT.R* $\rightarrow$ skip fi

Comp.*p*: if *v.p* $\neq \perp \rightarrow$ skip fi
 (*p* $\neq R$) *f.p* := *v.p*
 $(\forall q : n.q.p : s.p.q, v.q := \text{true}, p)$
 if *CT.p* $\rightarrow$ skip fi
t.p := *true*

The motivation for this change arises out of consideration of the target topology: each node is to terminate after a message from all its neighbours. Introducing variables *s.q.p* will allow us to test that *p* has received a message from all neighbours *q*. Shared variables *v.q* can now be eliminated

Inv: $T: \forall q :: q \neq R \Rightarrow t.q = \text{Comp}.q \text{ has terminated}$
 Pre: $\forall q :: f.q = \perp \wedge \neg t.q \wedge \forall p :: \neg s.p.q$

Comp.*R*: $(\forall q : n.q.R : s.R.q := \text{true})$
 if *CT.R* $\rightarrow$ skip fi

Comp.*p*: if $\exists q :: s.q.p \rightarrow$ skip fi
 (*p* $\neq R$) $(\exists q : s.q.p : f.p := q)$
 $(\forall q : n.q.p : s.p.q := \text{true})$
 if *CT.p* $\rightarrow$ skip fi
t.p := *true*

Next the private variable $t.q$ is to be replaced by $s.q.(f.q)$ according to the coupling invariant U , which requires strengthening the guard of action

$s.p.q := \text{true}$ with $q \neq f.p$

Inv: $T: \forall q :: q \neq R \Rightarrow t.q = \text{Comp}.q$ has terminated

$U: \forall q :: t.q = s.q.(f.q)$

Pre: $\forall q :: f.q = \perp \wedge \neg t.q \wedge \forall p :: \neg s.p.q$

Comp. R : $(\forall q : n.q.R : s.R.q := \text{true})$
 if $CT.R \rightarrow \text{skip}$ fi

Comp. p : if $\exists q :: s.q.p \rightarrow \text{skip}$ fi
 $(p \neq R)$ $(\exists q : s.q.p : f.p := q)$
 $(\forall q : n.q.p \wedge q \neq f.p : s.p.q := \text{true})$
 if $CT.p \rightarrow \text{skip}$ fi
 $s.p.(f.p), t.p := \text{true}, \text{true}$

While the replacement of $t.q$ by $s.q.(f.q)$ looks like a rabbit out of a hat, in fact, it arises out of consideration of the inference (*) to be arranged. If a node is to terminate after a message from all its neighbours and this is to imply that its children have terminated, we do well to arrange a message from a child to coincide with the child's termination. Private variables $t.q$ can now be eliminated

Inv: $T: \forall q :: q \neq R \Rightarrow s.q.(f.q) = \text{Comp}.q$ has terminated

Pre: $\forall q :: f.q = \perp \wedge \forall p :: \neg s.p.q$

Comp. R : $(\forall q : n.q.R : s.R.q := \text{true})$
 if $CT.R \rightarrow \text{skip}$ fi

Comp. p : if $\exists q :: s.q.p \rightarrow \text{skip}$ fi
 $(p \neq R)$ $(\exists q : s.q.p : f.p := q)$
 $(\forall q : n.q.p \wedge q \neq f.p : s.p.q := \text{true})$
 if $CT.p \rightarrow \text{skip}$ fi
 $s.p.(f.p) := \text{true}$

where guard $CT.p$ is true when p has received a message from all its children, which, given T , is when all p 's children have terminated

$CT.p : \forall q :: CT.q.p$

$CT.q.p : (n.q.p \wedge q \neq R) \Rightarrow (f.q \neq \perp \wedge (f.q = p \Rightarrow s.q.(f.q)))$

But recalling that the topology of the echo program is that each node terminates after a message from all of its neighbours (not all of its children), define $NR.p$ to be true when p has received a message from all of its neighbours

$NR.p : \forall q :: n.q.p \Rightarrow s.q.p$

A valuable simplification is now made by strengthening the $CT.p$ guard to $NR.p$, at the cost of a new program invariant S . (The simplification is valuable because variable $f.p$ is again made local to $\text{Comp}.p$.)

$$\begin{aligned}
 CT.p & \quad \Leftarrow \quad \{q \text{ is arbitrary.}\} \\
 CT.q.p & \quad = \\
 & (n.q.p \wedge q \neq R) \Rightarrow (f.q \neq \perp \wedge (f.q = p \Rightarrow s.q.(f.q))) \\
 & \quad = \\
 n.q.p & \Rightarrow (q \neq R \Rightarrow (f.q \neq \perp \wedge (f.q = p \Rightarrow s.q.(f.q)))) \\
 & \quad = \\
 n.q.p & \Rightarrow (q = R \vee (f.q \neq \perp \wedge (f.q \neq p \vee s.q.(f.q)))) \\
 & \quad \Leftarrow \quad \{s.q.p \Rightarrow (q = R \vee f.q \neq \perp) \text{ from } S\} \\
 n.q.p & \Rightarrow s.q.p \\
 & \quad \Leftarrow \\
 NR.p &
 \end{aligned}$$

The result is the echo program, where $NR.p$ is globally correct by the rule of widening, because $s.q.p$ is never falsified

2 (Echo program)

Inv: $T: \forall q :: q \neq R \Rightarrow s.q.(f.q) = \text{Comp}.q \text{ has terminated}$
 $S: \forall q, p :: (q \neq R \wedge f.q = \perp) \Rightarrow \neg s.q.p$
 Pre: $\forall q, p :: \neg s.q.p$

Comp. R : $(\forall q : n.q.R : s.R.q := \text{true})$
if $NR.R \rightarrow \text{skip}$ **fi** $\{NR.R\}\{CT.R\}$

Comp. p : **if** $\exists q :: s.q.p \rightarrow \text{skip}$ **fi**
 $(p \neq R) \quad (\exists q : s.q.p : f.p := q) \{f.p \neq \perp\}$
 $(\forall q : n.q.p \wedge q \neq f.p : s.p.q := \text{true})$
if $NR.p \rightarrow \text{skip}$ **fi** $\{NR.p\}\{CT.p\}$
 $s.p.(f.p) := \text{true}$

It remains to check Requirements (2) and (3), for which purpose it will be assumed that the first half of Program (2) computes a spanning tree. (This is proved in the postscript below.) Call this code STb . Under this assumption Program (2) meets Requirement (2). Comp. R also terminates (3).

Comp. R does not terminate
 $= \quad \{STb \text{ terminates}\}$
 Comp. R is blocked at $NR.R$ guard
 $=$
 $\neg s.q.R$ is stably true for some q
 $= \quad \{\text{Topology of Comp}.q\}$
 a child q of R does not terminate
 $= \quad \{\text{Same argument}\}$
 a child r of q does not terminate
 $= \quad \{f \text{ define a finite tree}\}$
 $false$

4.1. Conclusions from derivation 2

Initial effort at this exercise was spent in trying to analyse the echo program in [8](p194) in order to construct a verifying proof of its correctness in the theory of Owicki and Gries. Failure at this is one bit of subjective evidence that multiprograms are easier to derive than to verify. In this derivation Program (1) is used quite effectively as a stepping stone, and the key to this is the knowledge that, in the solution, the termination of a node is preceded by the receipt of a message from all of its neighbours. This knowledge paves the way for the coordinate transformation that replaces variables v and t by s and replaces guard $CT.p$ by $NR.p$. It is satisfying that important aspects of

the second derivation are automatic, such as the invariant T , which is a key to Safety Requirement (2), and the proof of the Progress Requirement (3) is almost a copy of that in Program (1).

The motivation for this exercise has already been discussed in Section 1. The state of science regarding the presentation/ justification/ verification/ derivation of distributed programs is not yet mature. These are either made operationally or axiomatically, where axiomatic methods require a precise programming language, a precise language for asserting properties of programs, and precise rules for deriving these properties. (But contrary to the opinion in [5], it is moot that operational presentations are more comprehensible. See, for example, the contrary argument in [3](p91).) The theory of Owicki and Gries is set apart from the methods in [5] because its programming model is much closer to an implementation language than are Petri nets. The logic is also simpler, but this comes at the cost of a lack of rules for deriving progress properties. Owicki and Gries is a theory of partial correctness only and, as we have seen, this means that the progress Requirement (3) of the PIF program is proved in an informal way. All the same, the proof is not less convincing because of it. While it has been shown how to encompass progress properties by extending the theory of Owicki and Gries with temporal logic [7], this author prefers the pragmatic approach of [3](p91), where progress properties are somewhat left to chance, and if this sounds shocking, the reader should review the progress that is made in [3] before dismissing this attitude as unscientific. Indeed, significant progress in developing techniques for engineering (deriving) multiprograms has occurred, in large part, because of a conscious decision to retain the simplicity of the original theory of Owicki and Gries.

4.2. Postscript on the spanning tree in program 2

A routine adaptation of [3](p303) shows that Program STb satisfies properties (4b) and (5b)

- (4b) If STb terminates variables f define a spanning tree.
 (5b) STb terminates.

STb

Inv: $P: \forall q, p :: s.q.p \Rightarrow q \text{ is red}$
 Pre: $\forall q, p :: \neg s.q.p$
 All nodes and edges are white

Comp. R : Redden R
 $(\forall q : n.q.R : \{R \text{ is red}\} s.R.q := true)$

Comp. p : if $\exists q :: s.q.p \rightarrow skip$ fi $\{q \text{ is red}\}$
 $(p \neq R) \quad (\exists q : s.q.p : f.p := q) \{f.p \text{ is red}\} \{p \text{ is white}\}$
 Redden p and edge $(f.p, p)$
 $(\forall q : n.q.p \wedge q \neq f.p : \{p \text{ is red}\} s.p.q := true)$

Nodes and edges are initially white and nodes and edges are reddened during the computation. Invariant P is locally correct by the precondition ' $_$ is red' of $s._.q$ assignments. Assertions ' $_$ is red' are globally correct by widening (nodes only turn red). Assertion ' p is white' is globally correct because only p can redder p .

Invariant to STb is that the red graph is a tree: every new edge uses a (white) node from outside of the red graph. If STb terminates then the tree is spanning because the network is connected. This is requirement (4b).

Requirement (5b) is true when all nodes are reddened and, because Comp. R is reddened and the network is connected, this is true when all neighbours of p are reddened after p is. Sometime after p is reddened $\forall q :: n.q.p \wedge q \neq f.p \Rightarrow s.p.q$ is true, which makes the guards of all neighbours of p , but $f.p$, true and stable. Thus all neighbours of p , but $f.p$, will be reddened, and $f.p$ is already red.

4.3. Postscript on the programming model and atomicity

In Section 1 the meaning of a multiprogram execution was given as a fair interleaving of the atomic actions of its components. In order to confirm the foregoing derivation it is now necessary to define what an atomic action is, and we shall do this, not by defining atomicity in terms of how program variables are used, but by abandoning this communication model altogether, for an asynchronous message passing model in which the atomic actions are transparent.

The first step is a syntactic transformation of Program (2), where assignment $s.p.q := true$ is replaced by the send action ' p sends to q ' and guarded command $\text{if } n.q.p \Rightarrow s.q.p \rightarrow skip \text{ fi}$ is replaced by the receive action ' p receives from q '

Comp. R : $(\forall q : n.q.R : \text{send to } q)$
 $(\forall q : n.q.R : \text{receive from } q)$

Comp. p : $\text{receive from } q$
 $(p \neq R) \quad f.p := q$
 $(\forall q : n.q.p \wedge q \neq f.p : \text{send to } q)$
 $(\forall q : n.q.p \wedge q \neq f.p : \text{receive from } q)$
 $\text{send to } f.p$

The syntax is adapted from [8](pp552-553). Program variables are now local to components which means that the atomicity of assignment and guard actions is no longer at issue. The atomic actions are send actions and receive actions. The send action must specify an argument which is the message destination component. The receive action must specify a parameter which, when the action is executed, is instantiated to the message source component. In this program the messages that are sent and received are simple signals. Since message passing is asynchronous, a send action is nonblocking, while a receive action blocks until a message is available.

The second step is to restore the program annotation

$NR.p : \quad \forall q :: n.q.p \Rightarrow q \text{ has sent to } p$

$CT.p : \quad \forall q :: (n.q.p \wedge q \neq R) \Rightarrow (f.q \neq \perp \wedge (f.q = p \Rightarrow q \text{ has sent to } f.q))$

3 (Echo program)

Inv: $T: \forall q :: q \neq R \Rightarrow (q \text{ has sent to } f.q = \text{Comp}.q \text{ has terminated})$
 $S: \forall q, p :: (q \neq R \wedge f.q = \perp) \Rightarrow \neg(q \text{ has sent to } p)$
Pre: $\forall q, p :: \neg(q \text{ has sent to } p)$

Comp. R : $(\forall q : n.q.R : \text{send to } q)$
 $(\forall q : n.q.R : \text{receive from } q) \{NR.R\}\{CT.R\}$

Comp. p : $\text{receive from } q$
 $(p \neq R) \quad f.p := q \{f.p \neq \perp\}$
 $(\forall q : n.q.p \wedge q \neq f.p : \text{send to } q)$
 $(\forall q : n.q.p \wedge q \neq f.p : \text{receive from } q) \{NR.p\}\{CT.p\}$
 $\text{send to } f.p$

REFERENCES

- [1] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [2] E. W. Dijkstra. A Personal Summary of the Gries-Owicki Theory, in *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982.
- [3] W. H. J. Feijen, A. J. M. van Gasteren. *On a Method of Multi-Programming*. Springer-Verlag, 1999.
- [4] D. Goldson. Multiprogram Design in the theory of Owicki and Gries, in *Proceedings of*

IWFM'01 5th Irish Workshop in Formal Methods. eWiC

(<http://www1.bcs.org.uk/>), 2001.

- [5] E. Kindler, W. Reisig, H. Volzer, and R. Walter. Petri Net Based Verification of Distributed Algorithms: An Example, in *Formal Aspects of Computing*. 9:4:409-121, 1997.
- [6] S. Owicki, D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach, in *Communications of the ACM*. Vol 19, No. 5, 1976.
- [7] F. B. Schneider. *On Concurrent Programming*. Springer-Verlag. 1997.
- [8] G. Tel. *Introduction to Distributed Algorithms*. Cambridge, 2000.