# Design and verification of distributed networks algorithms



*DOI:*
10.6100/IR328112

*Document status and date:*
Published: 01/01/1989

*Document Version:*
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

# Design and Verification
# of
# Distributed Network Algorithms:
# Foundations and Applications

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven,
op gezag van de Rector Magnificus, prof. ir. M. Tels,
voor een commissie aangewezen door het College van Dekanen
in het openbaar te verdedigen op
vrijdag 15 december 1989 te 16.00 uur

door
Frank Alwin Stomp
geboren te Gorssel

Dit proefschrift is goedgekeurd door

de promotoren

prof. dr. Willem-P. de Roever

en

prof. dr. Helmut A. Partsch.

Het onderzoek van Frank Stomp als beschreven in dit proefschrift
is verricht aan de Katholieke Universiteit te Nijmegen.

To my parents

CONTENTS

ACKNOWLEDGEMENTS

Willem-Paul de Roever has introduced me into the field of concurrency. I express my gratitude for his support, guidance, and constructive criticism during my research in this field. Many thanks are due to Corinne de Roever. She and Willem-Paul were always there for me when I needed it.

I would like to thank Helmut Partsch for his stimulation and for providing me the opportunity to finish this thesis.

Thanks also to Ralph Back and John-Jules Meyer for their willingness to referee this thesis.

Rob Gerth is thanked for being a co-author of one of the articles included in this thesis, and for his help during the last stage of writing this thesis.

I thank my parents for their continuous support. I am very grateful to Marilyn and Carla for their help on many occasions. The interest in my work shown by my brothers has always been appreciated.

Finally, I thank Myrthia for her understanding and support.

# CHAPTER 1

## Overview

This thesis collects four articles:

(1) F.A. Stomp and W.P. de Roever,

*A correctness proof of a distributed minimum-weight spanning tree algorithm* (*extended abstract*), which has been published in the proceedings of the 7th International Conference on Distributed Computing Systems, Eds. R. Popescu-Zeletin, G. Le Lann, and K.H. Kim. The full version of this article has appeared as technical report no. 87-4, University of Nijmegen, 1987.

(2) F.A. Stomp and W.P. de Roever,

*Designing distributed algorithms by means of formal sequentially phased reasoning.*

A version of this article has appeared as technical report no. 89-8, University of Nijmegen, 1989; An extended abstract has been published in the proceedings of the 3rd International Workshop on Distributed Algorithms, LNCS 392, Eds. J.-C. Bermond and M. Raynal.

(3) F.A. Stomp and W.P. de Roever,

*A detailed analysis of Gallager, Humblet, and Spira's distributed minimum-weight spanning tree algorithm – An example of sequentially phased reasoning –.*

(4) F.A. Stomp, W.P. de Roever, and R.T. Gerth,

*The μ-calculus as an assertion-language for fairness arguments.*

It has appeared in Information and Computation, Vol. 82, no. 3 (1989).

The central theme of the first three articles on *distributed program design and verification* is the identification, the technical formulation, and an application of a principle for designing, and verifying, (complex) distributed algorithms. This principle allows one to structure the design, or the verification, of algorithms from a certain class according to a particular pattern of reasoning.

This class consists of algorithms in which some group of nodes in a network performs a certain *task* which can be decomposed into a number of *subtasks* as if they are performed *sequentially* from a *logical* point of view. In reality, however, i.e., from an *operational* point of view, the subtasks are performed *concurrently*. A typical example in which one can discern this kind of sequential decomposition is Segall's PIF-protocol [S83]. The PIF-protocol, where "PIF" abbreviates Propagation of Information with Feedback, is a simple broadcasting protocol. All nodes in a finite, connected, and undirected network accomplish the following *task*: Some value initially recorded by a certain node $k$ is supplied to all nodes in the network and node $k$ is informed that all nodes in the network have recorded this value. This task can be decomposed into two *subtasks*: *the first one* broadcasting the value, and *the* *second one* reporting that the nodes have received and recorded this value.

The strategy proposed in the articles (1), (2), and (3) above to design (or verify) algorithms from the above-mentioned class is the following:

(a) First design algorithms which solve the subtasks. (This can be accomplished, e.g., by techniques advocated by Back and Sere [BS89] or by Chandy and Misra [CM88].)

(b) Then combine the algorithms found in (a) into one which solves the whole task.

This particular kind of strategy has been identified in (1).

The design principle formulated in (2), the second article of this thesis, describes how one could formally characterize the combination mentioned in (b) above.

This principle is applied in (3) to the complicated minimum-weight spanning tree algorithm of Gallager, Humblet, and Spira [GHS83].

The central theme of (4), the fourth article of this thesis, on *fairness arguments* is the formulation of an assertion expressing that a nondeterministic program terminates fairly. It is shown in (4) that this assertion can be formulated in Hitchcock and Park's monotone $\mu$-calculus [HP73]. This calculus is a formalism, based on Knaster and Tarski's fixed point theorem [T55], that can serve, as shown in (4), as an assertion-language for reasoning about fair termination of nondeterministic programs in a sound and (relatively) complete manner.

Meyer [M86] has used fixed points, too, for constructing a calculus that describes how to merge fairly operations of nondeterministic processes. An excellent overview on fairness issues has been given by Francez [F86].

Manna and Pnueli's Linear Time Temporal Logic [MP83], hereafter abbreviated to LTL, runs both in its applications and in its foundations, through the research reflected in all four articles like a thread. The design, hence, verification principle, which is the subject in (1), (2), and (3), is directly formulated using LTL. In the fourth article the foundations of LTL are investigated.

The results described in (1), (2), (3), and (4) are briefly sketched below.

In (1) it is sketched how the distributed minimum-weight spanning tree algorithm of Gallager, Humblet, and Spira [GHS83] can be proved to be correct. It is argued that the proof can be structured by *decomposing the reasoning* about the program describing that algorithm into a *number of loosely* *connected or independent arguments concerning distributed parts* of that program as if they are performed one after another. (In the terminology used above, the nodes which execute such a distributed part perform a certain subtask. The whole task consists of all these subtasks as if they are performed sequentially.) These distributed parts are *not syntactically* contained in the whole program. They are combinations of scattered pieces of text of various programs performed by the nodes, which *semantically* constitute a meaningful whole. It is claimed in (1) that the principle applied generalizes Elrad and Francez' principle of *communication closed layers* [EF82]. From the technical formulation of the principle in (2), it follows that it is a *broad semantic generalization* of Elrad and Francez' principle in that it is not restricted by the syntax of a programming language at all, whereas in Elrad and Francez' formulation the principle is restricted by the syntax.

Elrad and Francez' principle of communication closed layers [EF82] states the following:
Let $d \geq 1$ be some natural number. If for all $m$, $1 \leq m \leq d$, the programs $S_{1,m} \parallel \cdots \parallel S_{n,m}$, $n \geq 1$, are partially correct w.r.t. the preconditions $p_{m-1}$ and the postconditions $p_m$ and if no communication occurs between $S_{i,m}$ and $S_{j,m'}$ for $1 \leq i, j \leq n$, $i \neq j$, $1 \leq m, m' \leq d$, and $m \neq m'$ then, the program $(S_{1,1};S_{1,2};\cdots;S_{1,d}) \parallel \cdots \parallel (S_{n,1};S_{n,2};\cdots;S_{n,d})$ is partially correct w.r.t. precondition $p_0$ and postcondition $p_d$. (Here, as usual, program $S$ is partially correct w.r.t. precondition $p$ and postcondition $q$ if the following is satisfied: if $S$ is executed in an initial state satisfying $p$, then $q$ holds if and when $S$ terminates). The programs $S_{1,m} \parallel \cdots \parallel S_{n,m}$, $1 \leq m \leq d$, are called *layers* in [EF82].
This principle can be illustrated by means of the picture below. For ease of exposition, we consider the case of two layers. Let $\{p\}S\{q\}$ denote the assertion that the program $S$ is partially correct w.r.t. precondition $p$ and postcondition $q$. Elrad and Francez' principle asserts that if

$$\begin{array}{c} \{p_0\} \\ \boxed{S_{1,1} \parallel \cdots \parallel S_{i,1} \parallel \cdots \parallel S_{j,1} \parallel \cdots \parallel S_{n,1}} \\ \{p_1\} \end{array}$$

and

$$\begin{array}{c} \{p_1\} \\ \boxed{S_{1,2} \parallel \cdots \parallel S_{i,2} \parallel \cdots \parallel S_{j,2} \parallel \cdots \parallel S_{n,2}} \\ \{p_2\} \end{array}$$

both hold and if no communication occurs between $S_{i,1}$ and $S_{j,2}$ for all $i, j$ satisfying $1 \leq i, j \leq n$ and

$i \neq j$, then

$$\{p_0\}$$

$$\begin{array}{|c||c||c||c||c||c||c|}
\hline
S_{1,1}; & & S_{i,1}; & & S_{j,1}; & & S_{n,1}; \\
S_{1,2} & \ldots & S_{i,2} & \ldots & S_{j,2} & \ldots & S_{n,2} \\
\hline
\end{array}$$

$$\{p_2\}$$

is satisfied.

The principle which underlies the correctness proof in our paper (1) and which generalizes the principle of communication closed layers is, however, not explicitly formulated nor justified in (1) itself. (The proof suggested there should therefore be considered incomplete.)

In (2) the principle underlying the reasoning in (1) is formulated using LTL. This principle is applied in (3) to the minimum-weight spanning tree algorithm of Gallager, Humblet, and Spira, which is a representative of the class of algorithms we are interested in. In this algorithm following features occur:

- Tasks performed by groups of nodes in the network can be split up into a number of subtasks as if they are performed one after another from a logical point of view, although from an operational point of view they are performed concurrently.

  **Example:**

  This feature can be illustrated by the program below which describes the PIF-protocol in case the underlying network constitutes a tree. (This restriction is imposed in order to keep the presentation as simple as possible.) Recall that the PIF-protocol solves the following task: All nodes in a finite, connected, and undirected network are provided with some value initially recorded by a certain node $k$, and node $k$ is informed that all nodes in the network have recorded this value. Furthermore, recall that this task can be split up into two subtasks as if they are performed sequentially, the first one supplying all nodes in the network with the value to be propagated, and the second one reporting that all nodes have indeed received this value.

  In the program below, boxes labeled $A_i^n$ indicate which operations of node $i$ are associated with the $n^{th}$ subtask ($n$=1,2). Observe that boxes do not necessarily correspond to the body of a "response". (In general, such boxes are the outcome of a *semantic* analysis and not of a *syntactic* one.) Note that during the first subtask a directed tree is unwound. This tree is used by the nodes during the second subtask in order to trace their path back to node $k$ in order to inform $k$ that they have recorded the value which has been propagated.

**loop executed by node $k$ (the root)**

$A_k^1$

response to receipt of *info(v)*
begin
  $val_k := v$;
  for all edges $e \in E_k$
    do send $info(val_k)$ on edge $e$ od
end

$A_k^2$

response to receipt of *ack(v)* on edge $C$
begin
  $N_k(C) :=$ true;
  if $\forall C \in E_k . N_k(C)$
  then $done_k :=$ true
  fi
end

**loop executed by node $i \neq k$ (a non-root)**

$A_i^1$

response to receipt of *info(v)* on edge $C$
begin
  $val_i := v$; $inbranch_i := C$; $N_i(C) :=$ true;
  for all edges $e \in E_i \wedge e \neq inbranch_i$
    do send $info(val_i)$ on edge $e$ od;

  if $\forall C \in E_i . N_i(C)$
  then send $ack(val_i)$ on $inbranch_i$
  fi
end

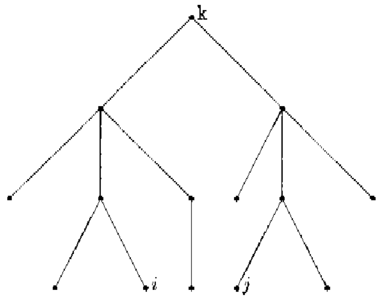
$A_i^2$

response to receipt of *ack(v)* on edge $C$
begin
  $N_i(C) :=$ true;
  if $\forall C \in E_i . N_i(C)$
  then send $ack(val_i)$ on $inbranch_i$
  fi
end

**Notation used:** $E_i$ denotes the set of edges adjacent to node $i$. Variable $val_i$ is used to record the argument of the info-message received by node $i$; $N_i(C)$ records whether any message has been received along edge $C$, $C \in E_i$. For node $i$ different from $k$, variable $inbranch_i$ records the identification of the edge along which the info-message has been received. (These variables are used for unwinding the directed tree.) Variable $done_k$ records whether the whole task has been completed. Each node maintains its own message queue for buffering received messages.
Initially, node $k$'s message queue contains one info-message and the message queues of all other nodes are empty. Furthermore initially $\neg done_k$ holds for node $k$, and $\neg N_i(C)$ for all nodes $i$ and edges $C \in E_i$. The initial values of the other variables are irrelevant.

Segall's PIF-protocol

From a logical point of view it seems as if first $A^1$ programs are executed (solving the first subtask) and thereafter only $A^2$ programs (solving the second subtask). Operationally, however, this kind of sequentialization is not necessarily true. This is exemplified below. Consider the following tree:




In general, obviously, the nodes $i$ and $j$ will not be supplied simultaneously with the value being propagated. There exist computation sequences of the program above for which the following is satisfied:

Node $i$ receives the value that is being propagated and records this value (node $i$ executes the program segment labeled $A_i^1$).

Then node $i$ enters the reporting phase (node $i$ executes the program segment labeled $A_i^2$).

Thereafter, node $j$ receives and records the message that is being propagated (node $j$ executes the program segment labeled $A_j^1$).

This example illustrates that the program segment $A_j^1$ is executed after node $i$ has executed the segment $A_i^2$, i.e., node $i$ participates in the second subtask before node $j$ participates in the first subtask.

Now, the principle formulated in (2) justifies that one can reason about the PIF-protocol as if first all $A^1$ programs are executed and thereafter only $A^2$ programs.

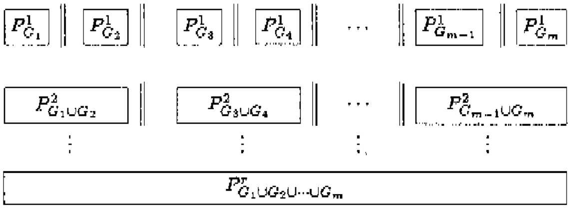
The next feature occurring in the distributed minimum-weight spanning tree algorithm of Gallager, Humblet, and Spira is the following (a principle for reasoning about this feature is formulated in (3)):

- *Expanding* groups of processes perform a certain task *repeatedly*, whereas different groups of nodes perform their task concurrently w.r.t. another.

E.g., the distributed minimum-weight spanning tree algorithm of Gallager, Humblet, and Spira can be described as follows:

First a certain collection of groups of nodes performs some task concurrently w.r.t. another. The task of each such group consists of determining the minimum-weight outgoing adjacent edge for any node in this group. Thereafter, a *fragment*, i.e., some subtree of the minimum-weight spanning tree to be constructed, which has determined its minimum-weight outgoing edge, attempts to combine with the fragment at the other end of this edge. The task of accomplishing this combination is then performed by all nodes in these two fragments. Subsequently, the enlarged fragment performs the task of determining its minimum-weight outgoing edge. This process is repeated until the minimum-weight spanning tree of the network has been constructed.

This feature is suggested in the following picture:

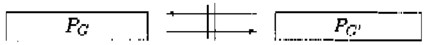



**Notation used:** For each $\ell$=1,$\cdots$,$r$, $P^\ell_G$ denotes a distributed program performed by nodes in a collection $G$. The superscripts are used only in order to distinguish the tasks associated with such programs; $r$ in the picture denotes some natural number, $r \geq 1$.
Initially, the collection consisting of $G_1, \cdots, G_m$ for some $m \geq 1$ is a partitioning of the set of all nodes in the network.

- A task performed by one group of processes can be disturbed temporarily due to *interference* with the task of another group.

In the distributed minimum-weight spanning tree algorithm of Gallager, Humblet, and Spira a fragment will, in order to determine its minimum-weight outgoing edge, send messages to nodes outside this fragment. This implies that a certain node in some group $G$ of nodes performing some task can receive messages from nodes outside this group which are not associated with the task in which the node itself participates. Consequently, when a node in $G$ receives a message not associated with the task in which its participates and it processes this message the task will be disturbed. After processing this message the node will continue its participation in the task.

Depicted in a picture, we have




**Notation used:** $P_G$ and $P_{G'}$ are distributed programs performed by nodes in group $G$ and $G'$ respectively. These programs are executed concurrently; Each of them describes how to solve a certain task. The arrow indicates the transmission of a message.

A principle which copes with the latter feature is formulated in (3). In essence, interference freedom of specifications has to be proved in order to ensure that the reasoning about the two tasks according to the principle, described in (2), is not invalidated.

Now, suppose that two distributed programs have been designed that solve two subtasks of a certain task as if they are performed sequentially. Assume that each of the subtasks and the task are described by means of a precondition and a postcondition. In order to design a program that solves the whole task it is required to prove that for each of the programs the following holds.
For each node $j$ that participates in the subtask, there exist for the program associated with this subtask when it is executed in an initial state satisfying the subtask's precondition:

An *invariant* $I_j$ which holds during execution of the program. These invariants have been incorporated in order to deal with the above-mentioned kind of interference. The invariant $I_j$ can be thought of as the disjunction of all predicates assigned to control points of the program when reasoning about this program in an Owicki-Gries-like proof system [OG76].
A *termination condition* $T_j$. $T_j$ holds when and if node $j$ has completed its participation in the program.

In addition, it must be proved that upon termination of the program the subtask's postcondition associated with this program is established, provided that execution has been started in a state satisfying the subtask's precondition.

A program which solves the whole task consists of all operations occurring in any of the programs solving the subtasks. This holds because a node participates in the whole task iff it participates in one of the subtasks. Furthermore, the following *verification conditions* must be shown to hold:

- A node can only participate in one subtask at a time.
  If a node actually participates in both subtasks, then it participates in the first subtask before it participates in the second subtask.

The first verification condition above ensures that there does not occur any communication between program segments associated with distinct subtasks. It also states that two internal operations (, i.e., operations not involving any communication), which can be performed by the same node and which are associated with distinct subtasks cannot be enabled simultaneously. The latter requirement is not needed in case of Elrad and Francez' principle, since it follows from the syntactic structure of the whole program. In case of their principle, the second verification condition above also follows from their condition about communication and from the syntactic structure of the whole program.

As mentioned above, the principle formulated in (2) is a generalization of the principle of communication closed layers.
The principle formulated in (2) also generalizes each of the principles formulated by Chou and Gafni [CG88], by Fix and Francez [FF89] and by Back and Sere [BS89], since, amongst others, none of these is able to cope with the above-mentioned kind of interference.

The principle formulated in (2) is applicable to the spanning tree algorithm of Gallager, Humblet, and Spira. This is shown in (3). As a consequence of the strategy adopted there, a source of failure of the algorithm has been detected and corrected. Also, two kinds of slight optimizations w.r.t. the number of messages transmitted during execution of the algorithm have been found.

At this stage the question might be asked why we did not apply a conventional proof system, such as described in, e.g., [AFR80, OG76] or [ZRE85], to prove the correctness of this algorithm. This question is answered below.
Apart from the algorithm reported in [GHS83], there exists a large number of algorithms [H83, MS79, S82, S83] of which the underlying structuring principle is inherently semantic. Despite the fact that the designers of such (complex) algorithms are able to give a clear and intuitive explanation about their correctness, it is believed that any correctness proof given in a conventional formalism can capture this intuition in an artificial way only. This implies that any such formal proof of a non-toy program will not contribute more to one's understanding of the designer's argument. The principle formulated in (2) is able to mimic the designer's argument in a straightforward manner, indeed.

In (4), the last article of this thesis, the foundations of LTL are investigated. This is done by studying the notion of strongly-fair termination of programs. In order to define this notion, the notion of a strongly-fair computation sequence is introduced: a computation sequence of a program is *strongly-*

*fair* if every operation occurring in the program which is infinitely often enabled in this sequence is infinitely often chosen in that sequence. Now, a program that is executed in an initial state satisfying some precondition $p$ *terminates strongly-fair*, if every strongly-fair computation sequence started in a state for which $p$ holds is finite.

E.g., Dijkstra's random number generator, see [D76], $*[b \rightarrow x := x+1 \ \Box \ b \rightarrow b := \mathit{false}]$ always terminates strongly-fair. This holds because of the following:

- The program immediately terminates when executed in a state satisfying $\neg b$.
- Any infinite computation sequence of the program started in a state satisfying $b$ is not strongly-fair, since this implies that the operation "$b \rightarrow b := \mathit{false}$" is infinitely often enabled and never taken.

Strongly-fair termination of a program is an example of an "eventually"-property when the above restrictions are imposed on computation sequences of the program. Manna and Pnueli [MP83] have presented a proof principle that allows one to establish such properties. They propose the following strategy to prove that for a program $S$, a state-property $\psi$ eventually holds (a state-property is a property of program states expressible without any temporal operators):

(A) Amongst the concurrent processes executing $S$ a distinction is made between those processes whose execution brings $\psi$ always nearer (in [MP83] such processes are called *helpful* processes), and those processes that do not, i.e., whose execution does not bring satisfaction of $\psi$ any nearer (such processes are called *steady* processes in Manna and Pnueli's terminology).

(B) It must be shown that, for every computation sequence of the program $S$, if a helpful process is systematically avoided, then (B1) or (B2) below is satisfied.

(B1) The sequence is infinite and does not satisfy the above fairness constraint, i.e., it is unfair.

(B2) Due to some choice of a steady process, satisfaction of $\psi$ is brought nearer or even $\psi$ is established.

In case (B1) the computation sequence is unfair, since infinitely often a helpful process is enabled but only finitely many times taken. In case (B2) $\psi$ has become less far away from satisfaction.

Upon closer inspection, part (B) above requires application of the same strategy to a syntactically simpler program than $S$: remove all helpful processes from $S$, and prove that eventually one of the following holds: (i) $\psi$, (ii) getting nearer to $\psi$, or (iii) a helpful process is enabled.

The technical formulation of Manna and Pnueli's principle is shown below. There the following notions have been used, see [MP83]:

Let $S \equiv S_1 \parallel \cdots \parallel S_n$ be some program, $n \geq 1$. Let $\phi$ and $\phi'$ be state-formulae.

- $S_i$ *leads from* $\phi$ *to* $\phi'$ when every transition in $S_i$ establishes $\phi'$ provided $\phi$ is satisfied before ($i=1,\cdots,n$).

  $S$ *leads from* $\phi$ *to* $\phi'$ when for all $i$, $1 \leq i \leq n$, $S_i$ leads from $\phi$ to $\phi'$.

The technical formulation of the above-mentioned strategy is as follows:

Let $\mathcal{M}=(A,\leq)$ be a well-founded structure. Let $\phi(a)$ be a parametrized state-formula over $A$, where $a$ intuitively expresses how far away establishing $\psi$ is. Let $h{:}A \rightarrow \{1,\cdots,n\}$ be a helpfulness function identifying for each $a \in A$ the helpful process $S_{h(a)}$ for states satisfying $\phi(a)$.

$$\frac{\begin{array}{l}\vdash S \text{ leads from } \phi(a) \text{ to } [\psi \vee (\exists\beta \leq a.\ \phi(\beta))] \\ \vdash S_{h(a)} \text{ leads from } \phi(a) \text{ to } [\psi \vee (\exists\beta < a.\ \phi(\beta))] \\ \vdash \phi(a) \Rightarrow \Diamond[\psi \vee (\exists\beta < a.\ \phi(\beta)) \vee \text{Enabled}(S_{h(a)})]\end{array}}{\vdash (\exists a.\phi(a)) \Rightarrow \Diamond\psi}$$

The soundness proof of this principle requires induction over well-founded sets. On the other hand, this principle is (semantically) complete, i.e., if $\Diamond\psi$ holds, then naive set theory can be used to establish its premises.

Manna and Pnueli, however, do not give any formalism in which one can establish the premise of their principle. In order to supply such a formalism, in (4) a principle is considered for proving strongly-fair termination of (sequential) nondeterministic do-loops. In this principle the same kinds of auxiliary quantities, i.e., the well-founded structure, a ranking predicate, and a helpfulness function can be discerned as occurring in Manna and Pnueli's principle.

The principle investigated, which is called Orna's rule in (4), is due to Grümberg, Francez, Makowski, and de Roever [GFMR81] and is as follows ($[p]S[q]$ denotes that program $S$ is totally correct w.r.t. precondition $p$ and postcondition $q$, i.e., whenever $S$ is executed in an initial state satisfying $p$, then $S$ always terminates and each final state satisfies $q$):

Let $\mathcal{M}=(A,\leq)$ be a well-founded structure. Let $\pi$:$A \rightarrow$(States$\rightarrow$\{*true*, *false*\}) be a predicate. Let $\psi$ be a state-predicate, and let for each $a \in A$, $a$ not minimal (as denoted by $a > 0$), be given pairwise disjoint sets $St_a$ and $D_a$, such that $D_a \neq \emptyset$ and $D_a \cup St_a=\{1,\cdots,n\}$:

$$\begin{array}{l}
\vdash [\pi(a) \wedge a > 0 \wedge b_j]S_j[\exists a' < a.\ \pi(a')] \text{ for all } j \in D_a \\
\vdash [\pi(a) \wedge a > 0 \wedge b_j]S_j[\exists a' \leq a.\ \pi(a')] \text{ for all } j \in St_a \\
\vdash [\pi(a) \wedge a > 0] * [\sqcup_{i \in St_a}(b_i \wedge \bigwedge_{j \in D_a} \neg b_j) \rightarrow S_i][true] \\
\vdash r \rightarrow \exists a.\ \pi(a) \\
\vdash (\pi(a) \wedge a > 0) \Rightarrow \bigvee_{i=1}^{n} b_i \\
\vdash \pi(0) \Rightarrow ((\bigwedge_{i=1}^{n} \neg b_i) \wedge \psi) \\
\hline
\vdash [r] * [\sqcap_{i=1}^{n} b_i \rightarrow S_i][\psi]
\end{array}$$

Note that $b_i \rightarrow S_i$ ($i$=1,$\cdots$,$n$) can be interpreted as state transitions. Also note that in this principle the assignment $a \rightarrow (D_a, St_a)$ for $a > 0$ generalizes the notion of a helpfulness function of Manna and Pnueli's principle. Consequently, the same kind of auxiliary quantities are required to apply the above two principles.

In (4) it is shown that Hitchcock and Park's monotone $\mu$-calculus [HP73, P69], based on fixed points, augmented with constants for all recursive ordinals can serve as an assertion language for reasoning about strongly-fair termination of do-loops. Soundness and completeness of the principle in [GFMR81] are proved. In particular, the weakest precondition for strongly-fair termination of a do-loop w.r.t. some postcondition is shown to be expressible in the $\mu$-calculus.

The results shed an interesting light on LTL. Wolper [W81] has observed that not all regular expressions can be expressed in LTL (in fact, LTL can only express a proper subset of the regular expressions, cf. [T81]). Obviously, the $\mu$-calculus is far more expressive than the regular expressions. Consequently, in (4) a more expressive formalism than LTL has been used in order to express the auxiliary quantities required to apply the principle above. Although it has not been *proved* that one actually needs an assertion language at least as expressive as the $\mu$-calculus for reasoning about strongly-fair termination of do-loops –to my knowledge this is still an open problem– the results in (4) suggest strongly that one actually needs a formalism which is at least as expressive as LTL in order to formulate and verify the premises of Manna and Pnueli's principle mentioned above. To put it bluntly: *an "obvious" subformalism which Manna and Pnueli use in [MP83] to express their proof rules is probably more powerful than the whole of the LTL superstructure erected on top of that subformalism.*

## References


[AFR80] Apt K.R., Francez N., and de Roever W.P., A proof system for communicating sequential processes, ACM TOPLAS, 2-3 (1980).

[BS89] Back R.J.R. and Sere K., Stepwise refinement of action systems, LNCS 375 (1989).

[CG88] Chou C.T. and Gafni E., Understanding and verifying distributed algorithms using stratified decomposition, Proc. of the ACM Symp. on Principles of Distr. Comp. (1988).

[CM88] Chandy K.M. and Misra J., Parallel program design: a foundation, Addison-Wesley Publishing Company, Inc. (1988).

[D76] Dijkstra E.W., A discipline of programming, Prentice-Hall, Englewood Cliffs, NJ (1976).

[EF82] Elrad T. and Francez N., Decomposition of distributed programs into communication closed layers, Science of Computer programming, 2 (1982).

[F86] Francez N., Fairness, Springer Verlag (1986).

[FF89] Fix L. and Francez N., Semantics-driven decompositions for the verification of distributed programs, manuscript (1989).

[GFMR81] Grümberg O., Francez N., Makowsky J.A., and de Roever W.P., A proof rule for fair termination of guarded commands, Proc. Symp. on Algorithmic Languages (1981).

[GHS83] Gallager R.T., Humblet P.A., and Spira P.M., A distributed algorithm for minimum-weight spanning trees, ACM TOPLAS, 5-1 (1983).

[HP73] Hitchcock P. and Park D., Induction rules and termination, Proc. ICALP I, North-Holland (1973).

[H83] Humblet P.A., A distributed algorithm for minimum-weight directed spanning trees, IEEE Trans. on Comm., 31-6 (1983).

[M86] Meyer J.-J. Ch., Merging regular processes by means of fixed point theory, TCS 45, (1986).

[MP83] Manna Z. and Pnueli A., Verification of concurrent programs: A temporal proof system, Foundations of computer science IV, part 2, MC-tracts 159 (1983).

[MS79] Merlin P.M. and Segall A., A failsafe distributed routing protocol, IEEE Trans. on Comm., 27-9 (1979).

[OG76] Owicki S.S. and Gries D., An axiomatic proof technique for parallel programs, Acta Informatica 6 (1976).

[P69] Park D., Fixed point induction and proofs of program properties, Machine Intelligence 5 (1969).

[S82] Segall A., Decentralized maximum-flow algorithms, Networks 12 (1982).

[S83] Segall A., Distributed network protocols, IEEE Trans. on Inf. Theory. IT29-1 (1983).

[T55] Tarski, A., A lattice-theoretical fixed point theorem and its applications, Pacific J. Math. (5), (1955).

[T81] Thomas W., A combinatorial approach to the theory of $\omega$-automata, Information and Control 48 (1981).

[W81] Wolper P., Temporal logic can be more expressive, FCS (1981).

[ZRE85] Zwiers J., de Roever W.P., and van Emde Boas P., Compositionality and concurrent networks: soundness and completeness of a proof system, LNCS 194 (1985).

[ZS80] Zerbib F.B.M. and Segall A., A distributed shortest path protocol, Internal Report EE-395, Technion-Israel Institute of Technology, Haifa, Israel (1980).

# CHAPTER 2



Reprinted, with permission, from *Proceedings of the 7th International Conference on Distributed Computing Systems*, Berlin, West Germany, September 21-25, 1987, pp. 440-447.

# A correctness proof of a distributed minimum-weight spanning tree algorithm (extended abstract)

F.A. Stomp*

University of Nijmegen

W.P. de Roever**

University of Eindhoven

Abstract

We discuss a strategy to reduce the complexity of correctness proofs for particular classes of distributed programs. As an example of this strategy we sketch how a correctness proof of the distributed minimum-weight spanning tree algorithm of Gallager, Humblet, and Spira [7] can be structured by first introducing, and analyzing, simplifications in which certain communications are ignored. Then these simplifications are justified for the general case by proving that these communications do not affect the original analysis which is based on those simplifications. This proof - a more elaborated version of it can be found in the full paper [21] - illustrates the notions of *communication closed layer* of Elrad and Francez's [5] and of *quiescence* of, e.g., Chandy and Misra's [3].

## 1. Introduction

In order to reason about distributed programs, a number of methods have been proposed (e.g., [1,2,12,16,22]). While this enables an analyzer to verify that a program meets its specification, it turns out that these methods give, in general, rise to lengthy proofs for rather "simple" programs. (See [20] for an overview of some of these methods and an application of each of them to that ged of protocol-verification, the alternating bit protocol.) This suggests that correctness proofs of larger distributed programs are difficult to seize on. Consequently, the question arises, *whether there exist strategies which can reduce the complexity of proofs for particular classes of distibuted programs.*

The leitmotiv of this paper is the decomposition of the reasoning about a distributed program *into a number of loosely connected or independent arguments concerning distributed parts of that program under simplifying assumptions.* Typically, these distributed parts are not syntactically contained in one process but are combinations of scattered pieces of the text of various processes which constitute together a semantically meaningful whole. Equally characteristic for our approach is that we first reason in a simplified fashion about those distributed parts, disregarding interference due to certain communications from outside those parts, to argue later that in case these communications are taken into account, our reasoning remains valid, or can easily be adjusted to the "full reality" of interaction.

The suggested technique is an analogon of some techniques already suggested in the literature, such as Lam and Shankar's *method of projections*, Martin's analysis of the termination behavior of a distributed program using *quiescent states*, Chandy and Misra's method of *quiescence detection*, Lamport's argument that *reasoning about distributed programs* need not involve the constituting parallel processes or entities to base one's proof upon but *should be based rather on properties derived from global invariants*, and Elrad and Francez's technique of *communication closed layers*. These methods are briefly reviewed in the next section. The particular combinations of techniques used is illustrated by sketching a correctness proof of the distributed minimum-weight spanning tree algorithm of Gallager, Humblet, and Spira [7]. A more elaborated version of this proof can be found in [21].

Lam and Shankar [14] have proposed a technique of reducing the complexity of both safety and liveness properties of protocols. Their key observation is that protocols are, in general, designed to perform a number of distinct functions. E.g., in a communication protocol to achieve full-duplex data transfer between two stations one can discern two distinguishable functions, each one concerned with one-way data transfer between the two stations. To reduce the complexity of proofs, Lam and Shankar's technique decomposes such a multi-function protocol into a number of so-called image-protocols. Since these image-protocols perform, in general, less functions than the original one, they are easier to analyze. In [14] it has been proved that under certain conditions, safety and liveness properties verified for an image-protocol, carry over to the original one. To our knowledge, this method has been applied so far to communication protocols only (e.g., in [19]) and it appears that the applicability of the method to other classes of protocols remains open.

Martin presents in [15] a general technique to show termination of a distributed program. This technique consists of first deriving a non-terminating analogous program, for which it is proved that it reaches a state in which all internal activity has ceased and all channels in the network are empty, a so-called *quiescent* state. Next, a local termination condition from conditions satisfied in the quiescent state is derived which carries over to the original program. Although this technique reduces in some sense the complexity of a termination-proof it does not reduce the complexity of proofs of other properties.

More recently, Chandy and Misra have proposed a technique for the development of programs (see [3]). In their view, a program consists of an initial condition and a set of atomic actions. One of the key features of their methodology is that concerns about the core problem to be solved are separated from the forms of concurrency available in the hardware (on which the program is to be executed) and from the language in

---

* Department of Computer Science, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands. Electronic mail address: mcvax!kunivv1!kunivv5!stomp. The work of the first author was supported by the Netherlands Organization for the Advancement of Pure Research (ZWO).

** Department of Computer Science and Mathematics, POB 513, 5600 MB Eindhoven, The Netherlands. Electronic mail address: mcvax!eutrc3!wsinwpr.

CH2439-8/87/0000/0440$01.00 © 1987 IEEE

which the program is to be written. In [3] it has been shown that modeling a program as a set of statements is attractive, since it allows one to develop pieces of a program given only one invariant, independent of the other pieces of that program. This enables one to concentrate solely at one concern at a time.

In [3] a global view of the system under consideration is adopted. Although a number of authors have advocated compositional proofs (e.g., in [16,22]), Lamport has shown that assertional methods (involving reasoning about the global program states) are well-suited to reason about distributed programs, since they are not limited to the syntactic decomposition of a program into parallel processes (as compositional methods are), but also apply to decompositions which do not follow the syntactic decomposition. Lamport has illustrated this in [13].

Interesting in the field of program-verification is also the notion of communication closed layers as introduced by Elrad and Francez in [5]. (Subsequently, this notion has been deepened in [8].) One of the main aspects of communication closed layers is the simplification of the analysis of distributed programs by, again, suggesting a decomposition of a program consisting of parallel processes which runs across the syntactic boundaries of parallel decomposition by identifying groups of syntactic layers in the text of those processes which communicate exclusively which each other. Using CSP [9] to illustrate this notion, any process $P_i$ in $P \equiv [P_1 \| ... \| P_n]$ is represented as a sequential composition $S_{i,1};...;S_{i,d}$ for some d (i=1,...,n); d can be chosen uniformly by allowing $S_{i,j}$ to be an empty statement. For j=1,...,d, $L_j \equiv [S_{1,j} \| ... \| S_{n,j}]$ is called the $j^{th}$ *layer* of P. Layer $L_j$ is said to be *communication closed* iff under no execution of P, synchronization occurs between a communication command in some $S_{i,j}$ and a communication command in some $S_{k,j'}$ with $j \neq j'$. (In the terminology of [1], this is rephrased as follows: for any communication command in layer $L_j$ syntactically matching with a communication command in another layer, no semantical match will ever occur between them.) The decomposition of P into layers $L_1,...,L_d$ is a process $P' \equiv L_1;...;L_d$. Such a decomposition is called *safe* iff all the layers are communication closed. The relevance of such a decomposition of P is the following: let $P' \equiv L_1;...;L_d$ be a safe decomposition of P into layers $L_1,...,L_d$. Denote by $\{p'\}S'\{q'\}$ the assertion that S' is partially correct w.r.t. the precondition p' and the postcondition q', i.e., if S' is executed in an initial state satisfying p' and S' terminates, then the final state satisfies q'. Then

$$\frac{\{p_0\}L_1\{p_1\},...,\{p_{d-1}\}L_d\{p_d\}}{\{p_0\}P\{p_d\}}$$

is sound, and constitutes a derived proof rule in the system of [1]. Thus, under a suitable decomposition of P into layers, it suffices to verify the correctness of each layer first, and then establish the correctness of P by applying the above rule. In case communication is asynchronous, a safe decomposition is one in which sending and processing a message (contained in a message queue) takes place in the same layer. Observe that if $[S_{1,j} \| ... \| S_{n,j}]$ is a layer of a process $[P_1 \| ... \| P_n]$, then $S_{i,j}$ is syntactically contained in $P_i$ for i=1,...,n. One of the main contributions of this paper is a semantical generalization of the notion of communication closed layers.

In this paper, the distributed minimum-weight spanning tree algorithm of Gallager, Humblet, and Spira [7] serves as an example to illustrate how a correctness proof of a distributed program can be simplified by first introducing an abstraction from operational reasoning that certain communications can be ignored (at this stage of the proof). To express syntactically that certain communications are ignored, we replace the send-actions corresponding to those communications by **skip** in the program text. To simplify the reasoning, also a number of boolean conditions will be replaced by a constant boolean condition, i.e., either by **true** or by **false**. As we can apply Martin's technique [15] to analyze the termination behavior of a program, it follows that our technique also illustrates a generalization of the one presented in [15]. Secondly, this abstraction is used to decompose the program (whose execution is thus simplified) into a number of communication closed layers, whose existence would not have been justified without this abstraction. Thirdly, the program thus simplified is verified. Finally, our abstraction from operational reasoning is justified by demonstrating that the above assumptions can be eliminated, indeed, without invalidating our earlier proofs. Since the notions of quiescence and of communication closed layer play a rather significant role in this paper, it follows that we have put together a number of ingredients derived from some of the methods discussed in the previous section. Also, we have chosen to adopt Lamport's global view of a system.

Next, we summarize the main contributions of this paper.

(a) The notion of (communication closed) layers has been extended to the case of asynchronous communication.
(b) Application - to our knowledge for the first time - of (communication closed) layers in the field of protocol verification.
(c) The technique described in [15] has been generalized. Thus our technique does not only reduce the complexity of a termination-proof; it also enables an analyzer of such programs to reduce the complexity of other properties of them.
(d) Although no *safe* decomposition can be found for the program S embodying Gallager's algorithm, yet such a decomposition can be obtained after applying a *suitable abstraction* of the kind discussed above.
(e) In spite of the clear informal description in [7], it is far from being obvious that the formal description of the algorithm, i.e., the program S captures indeed those informal ideas. More precisely, the correctness of S has *not* been proved in [7], and there are a number of statements in S, such as conditionals, whose role has *not* been explained at all in that paper. E.g., consider the test whether a node should awaken, or whether a node should reject an edge in [7]. In the full paper [21], we have proved S's correctness. Also, we have shown there that the statements as mentioned above are of vital importance for its correctness. Moreover, we have given a formal justification of the (informal) reasoning in [7] and a slight optimization of that algorithm.

(d) and (e) above also motivates the choice of Gallager's algorithm to illustrate our verification technique.

The remainder of this paper is organized as follows. In chapter 2, we briefly review a number of properties known from graph-theory, that are essential to establish the correctness of Gallager's algorithm. In that chapter we also describe the skeleton of this algorithm. In chapter 3, we discuss the basic features of Gallager's algorithm. In that chapter, we also outline how S's correctness has been established in [21], and illustrate a decomposition of S to reduce the complexity of suc a correctness proof. (Here S denotes the program embodying Gallager's algorithm.) This decomposition illustrates a semantical generalization of Elrad and Francez's notion of communication closed layer [5]; the proof illustrates a generalization of Martin's technique [15]. Finally, chapter 4 contains the conclusion.

## 2. Preliminaries

We assume the reader to be familiar with the elementary definitions and properties of graphs, trees, paths, cycles, and so forth, which can be found in [6]. In particular, for graphs (V',E') and (V,E), (V',E') is a subgraph of (V,E), denoted by $(V',E') \subseteq (V,E)$, *iff* $V' \subseteq V$ and $E' \subseteq E$. If $(V',E') \subseteq (V,E)$ holds and moreover (V',E') is a tree, then (V',E') is called a subtree of (V,E). In the first section of this chapter, we will formulate a number of properties - well-known from graph theory - that are essential to establish the correctness of Gallager's algorithm. Because of the space limitations their proofs have been omitted. Thereafter, the skeleton for Gallager's algorithm is introduced and the model of computation is described.

Throughout this paper, (V,E) denotes a finite, undirected, and connected graph, where V is a set of nodes, and E is a set of edges. For $i \in V$, we denote the set of edges adjacent to i by $E_i$. Similarly, the set of edges adjacent to $i,j \in V$ is denoted by $E_{i,j}$. We assume each edge $e \in E$ has some weight $w(e)>0$ associated with it, such that different edges have different weights. The assumption that different edges have different weights implies that one can identify edges by their weights. Although one could relax this assumption somewhat, it is crucial for the correctness of Gallager's algorithm.

At the basis of Gallager's algorithm are the existence and the uniqueness of a minimum-weight spanning tree of any (V,E).

**Theorem 2.1**
Let $w:E \to \mathbb{R}^+$ be a function assigning weights to edges of (V,E), where $\mathbb{R}^+$ denotes the set of all real numbers greater than 0. (w is also referred to as the weight-function of the graph (V,E).) Assume that w is an injection. Then there exists a unique minimum-weight spanning tree of (V,E). []

Given some (V,E) and an injective w as above, theorem 2.1 ensures the existence of a unique minimum-weight spanning tree T. Throughout this paper, T always refers to this spanning tree of (V,E). A (naive) method to obtain this tree is the following: generate all spanning trees of (V,E) and determine the one with the minimum-weight among them. This requires a strategy to generate the spanning trees of (V,E). Another approach is suggested by theorem 2.2 below. Before formulating this theorem, we first introduce the notion of a fragment of T, and the notion of an outgoing edge of a fragment of T.

**Definition 2.1**
Given (V,E) and w as above. Denote by T the minimum-weight spanning tree of (V,E).
(a) A *fragment* of T is any non-empty subtree of T.
(b) Let T'=(V',E') be a fragment of T. An edge $e \in E$ is said to be an *outgoing edge* of T' iff one of the nodes adjacent to e is in V' and the other one is not. Consequently, edge e is an outgoing edge of T' iff $(i \in V' \wedge j \notin V') \vee (i \notin V' \wedge j \in V')$ holds, where i and j are nodes satisfying $e \in E_{i,j}$. []

We then have the following

**Theorem 2.2**
Let $T_k=(V_k,E_k)$, k=1,2, be fragments of T.
(a) Assume that $e \in E$ is the minimum-weight outgoing edge of $T_1$ and that e is adjacent to $T_2$ (i.e., adjacent to some node in $T_2$). Then $T_3=(V_1 \cup V_2, E_1 \cup E_2 \cup \{e\})$ is a fragment of T, too.
(b) $T=T_1$ iff there does not exist an outgoing edge of $T_1$. []

A large number of algorithms (e.g., [4,7,23,11]) have been suggested by theorem 2.2. Using this principle, one starts with the trivial fragments of T consisting of one node and no edges. To enlarge fragments, one or more fragments find their minimum-weight outgoing edge, if any. When (and if) such an edge has been found, the fragments on both sides of this edge may then be combined into one as described in theorem 2.2. This strategy ensures that fragments are constructed indeed. It also describes how fragments are enlarged. If, on the other hand, a fragment has no outgoing edges, then theorem 2.2 ensures that the fragment *is* the minimum-weight spanning tree of the graph.

The algorithms mentioned above differ in how and when fragments are enlarged. E.g., the algorithm reported in [4,23] starts with a single node as a fragment and gradually enlarges this fragment by appending the minimum-weight outgoing edge and the node adjacent to this edge, until the minimum-weight spanning tree T has been constructed. As such, constructing T is restricted to a rather strong requirement, not taking into account that many fragments could be combined into larger ones asynchronously from each other. In fact, this algorithm is inherently sequential. The algorithm reported in [11], however, starts with all fragments consisting of one node and no edges, and combines fragments into larger ones if they have the same minimum-weight outgoing edge. Thus, different fragments could be combined asynchronously from each other. Yet, fragments combine only, if they have the same minimum-weight outgoing edge.
Gallager's algorithm [7] starts with all fragments consisting of one node and no edges. Combining fragments into larger ones depends on their so-called levels. More precisely, fragments consisting of a single node are defined to be at level 0. Next, suppose that F is a fragment at level L with minimum-weight outgoing edge e. Let F' denote the fragment, say at level L', at the other end of e. If L=L' and e is F' 's minimum-weight outgoing edge, too, then they are combined into a larger fragment at level L+1(=L'+1). If L<L', then the fragments F and F' are combined into one at level L'. In all other cases, F has to wait until one of the two possibilities described above, occurs.

Above, we described the *skeleton* for Gallager's algorithm. It can be shown that the delay introduced in the skeleton (, hence in the algorithm,) does not lead to a deadlock, i.e., if a fragment waits for one of the conditions to combine with an other fragment into a larger one, then one of these conditions shall eventually occur.
Thus, in Gallager's algorithm many fragments can be combined into larger ones asynchronously from each other. Moreover two fragments may combine into a larger one regardless of whether they have identical minimum-weight outgoing edges.
Therefore, compared with the other algorithms mentioned

before, a "faster" algorithm has been yielded. Gallager's algorithm is a distributed one. Since there exist no global tables, messages have to be sent over edges to determine the minimum-weight outgoing edge of a fragment. Thus, if at some point during the algorithm fragment F has been constructed, each node in F should start searching for the *minimum-weight outgoing adjacent edge* by sending messages. Thereafter, cooperation must take place between all nodes in F to determine *the minimum-weight outgoing edge of F itself.* Observe that in order to determine whether an adjacent edge e of some node in F is actually an outgoing one of F, it suffices to determine whether the node at the other end of e belongs to F, too. Clearly, this is a difficult task, since the only way to find out whether two nodes belong to the same fragment is by means of sending messages. In Gallager's algorithm, nodes send so-called Test-messages on edges when searching for the minimum-weight outgoing edges. Without additional information, however, it is impossible to determine whether two neighbors belong to the same fragment. Thus, when nodes in a fragment start searching for their minimum-weight outgoing edge, they are all provided with a *name* of the fragment. This name enables adjacent nodes to determine whether they belong to the same fragment. Thus, when a node transmits a Test-message, this message also carries the name of its fragment as an argument. The receiver of the message informs the sender whether they have the same name. If so, the edge connects two nodes in the same fragment; otherwise the nodes adjacent to that edge belong to the different fragments. *Although this reasoning might suggest that it solves the problem of determining whether edges are outgoing, it does not.* The reason is that a node receiving a Test-message might have another name than the sender of the message, while both belong to the same fragment. This possibility occurs, if the receiver of the Test-message has not yet received the new name of its fragment. In [7], each Test-message carries an additional argument - the level of its sender's fragment - to avoid such undesired situations, i.e., situations in which an edge would have got the status of outgoing, while it is not. Introducing the levels has an other advantage, too, viz., it reduces the number of messages required to construct the minimum-weight spanning tree T (see [7]). In the next chapter, we will describe Gallager's algorithm in some more detail.

In the remainder of this chapter we describe our model of computation. This is done rather informally. The point of departure is a computer network (V,E), where V is a (finite) set of computing units, also referred to as nodes, and where E is a (finite) set of undirected communication channels, also referred to as edges. In the remainder of this paper, we assume that the network has a fixed topology. (The reader interested in algorithms that cope with failures and additions of edges or nodes, is referred to e.g., [17]). Additionally, we assume that the network is connected, and that each channel in the network connects exactly two distinct nodes. The latter assumption is important for the correctness of Gallager's algorithm. Consequently, such a computer network can be viewed as a finite, undirected, and connected graph.

The nodes in the network are assumed to possess a certain memory- and computation capability, and to be able to communicate via messages with their neighbors. Note that each node is able to transmit and receive messages on any channel adjacent to that node, since the channels are undirected. Messages transmitted by some node on a channel arrive within a finite, (but unpredictable) time-duration, in sequence, error-free, and without duplication at the other end of the channel.

The algorithm presented in the next chapter is distributed in the sense that no central tables are required and that there is no global knowledge of the topology. Each node "knows" only its adjacent channels and their weights. Each node is responsible for updating its own, i.e., local, tables and variables. The algorithm is such that all nodes obey the same local algorithm. At each node $i \in V$, there exists a program $S_i$ to perform its local algorithm. Variables occurring in $S_i$ are assumed to be subscripted by i. If no confusion can occur, then we omit these subscripts.

Transmitting a message M on an edge e can be achieved by executing a statement "send M on edge e". Each node maintains a message-queue. Upon receipt of a message, it is stamped with an identification of the edge on which it has been received. Each message-queue is supposed to work on a FIFO-basis. If a node's queue is non-empty, then the front message may be removed from its queue and either processed, or, as we will see, placed at the end of the queue, waiting for other events to occur. We assume that each queue's capacity is large enough to buffer all received messages. It is not difficult to derive a minimum size, such that each queue is able to buffer all received messages. This is not the subject of this paper, however.
In the sequel, we use the notation $queue_i$ to denote i's message-queue ($i \in V$). Also, we adopt the convention to denote e's contents of messages incoming to i by $contents_i(e)$ ($i \in V$, $e \in E_i$). Thus, for $i \in V$, $e \in E_i$, $contents_i(e)$ denotes the sequence of messages that has been transmitted by the other node adjacent to e, which has not yet been received by i.

We next fix some network (V,E) as described above, together with an injective weight-function $w: E \rightarrow \mathbb{R}^+$. One might view the weights w(e), $e \in E$, as the cost of transmitting a message on edge e.

## 3. A specification and an algorithm

In chapter 2, we have discussed the skeleton for Gallager's algorithm. In this chapter we are going to refine this skeleton somewhat. The ultimate goal is, of course, to show that Gallager's algorithm meets its specification. Therefore, we formulate a specification for a (distributed) program S that embodies the algorithm. In order to prove S's total correctness, i.e., if S is executed in an initial state satisfying some precondition, then S always terminates and in the final state the minimum-weight spanning tree T of (V,E) has been constructed, it suffices to show that each fragment finds its minimum-weight outgoing edge indeed and that fragments combine as described in theorem 2.2. This is established by induction on the level of a fragment (see [21]).

Now, a correctness proof of any complex distributed program should somehow be structured. It is convenient to structure the proofs reflecting the considerations of the (algorithm-)design. This observation has lead to decompose the program S embodying Gallager's algorithm into layers, thereby enabling the proof strategy described in chapter 1. One of the main advantages of this strategy is that proofs can be given, concentrating on one part of the program at a time. As an example of this, we mention an algorithm which is not identical to Gallager's algorithm, but captures the most essential features of Gallager's algorithm.

In the previous chapter we have discussed the skeleton for Gallager's algorithm. There we have also outlined the need for fragment's names. With this in mind, Gallager's algorithm can next be described as follows:

(a) A fragment at level 0, i.e., a fragment consisting of one node only, finds its minimum-weight outgoing edge according to its local information (, since any adjacent edge of such a node is an outgoing one). After finding this edge a Connect-message is transmitted on this edge. This message serves as a request of the fragment to combine with the fragment at the other end of that edge. This part of the program is performed by node i when executing $S_{i,1}$ in program S1 below.

(b) (i) If two fragments F and F' have found that they are at the same level L and that they have the same minimum-weight outgoing edge, then they are combined into one at level L+1. Each node in this newly formed fragment is then provided with a name and with the new level of this fragment. Node i participates in this part of the algorithm when executing $S_{i,2}$ in the program S1 below.

(ii) After receiving this name and level, the node starts searching for *its minimum-weight outgoing adjacent edge*, if any. If the nodes have ended this search, they should all cooperate to determine the edge with the least weight amongst all outgoing ones, if any. If there are no outgoing edges, then the algorithm terminates, since the minimum-weight spanning tree has been constructed (see theorem 2.2).
Node i participates in this part of the algorithm when executing $S_{i,3}$ in S1. Observe that $S_{i,3}$ is not syntactically contained in the program executed by node i. Yet, we have shown in [21] that the decomposition as illustrated in the program S1 is semantically meaningful. We have been able to prove that this decomposition induces layers which are communication closed after a number of simplifying assumptions. In the discussion after the program S1 below, we comment on these assumptions and their impact on the communication closedness of the layers.

(iii) If the minimum-weight outgoing edge of the fragment has been found in (ii) above, then the node in the fragment adjacent to this edge will be informed to send some Connect-message on this edge. This message serves as a request to combine with the fragment at the other end of this edge. Node i participates in this part of the algorithm when executing $S_{i,4}$ in S1.

(c) If a fragment F at level L has found its minimum-weight outgoing edge and the fragment F' at the other end of this edge is at level L' with L'>L, then F is immediately absorbed by F'. The new fragment is at level L'. This part of the algorithm has not been incoorpored in S1 below. In fact, these combinations ensure the progress in the algorithm, i.e., they ensure that the algorithm is deadlock-free.

(d) If a fragment F has found its minimum-weight outgoing edge e and none of the possibilities above is applicable, then F has to wait for combining with the fragment F' at the other end of e. In fact, this can occur in Gallager's algorithm only, if F and F' are at the same level and the following holds: F has not found its minimum-weight outgoing edge yet, or F' has a minimum-weight outgoing edge other than e.

With (V,E) and an injective weight-function $w:E \to \mathbb{R}^+$ as before, let T denote the graph's minimum-weight spanning tree (existing by theorem 2.1).
To give a specification for S, the program embodying Gallager's algorithm, we note that each node maintains its own variables to perform its part of S. One variable, $sn_i$, records the (node-)status of node i. Each node can be in one of the following states:
- *sleeping*, if it is not participating in the algorithm (yet),
- *find*, while it is participating in a fragment's search for determining the minimum-weight outgoing edge of the fragment,
- *found*, in all other cases.
Initially, each node in the network is in the sleeping state, i.e., no node participates in the algorithm.

Each node of the network also records the status of its adjacent edges, marking an adjacent edge as a
- *branch*, if the node has determined that the edge is in T,
- *rejected*, if the node has determined that the edge is not in T, or
- *basic*, in all other cases, i.e., if the node has not yet determined whether that edge is in T.
Each node $i \in V$ maintains a variable $se_i(e)$ to record the status of edge e ($e \in E_i$). We assume that initially each node has marked its adjacent edges as basic, i.e., we assume that initially $\forall i \in V\ \forall e \in E_i.\ se_i(e)=\text{basic}$ holds.
Consequently, initially no node participates in the algorithm, and each node is "unaware" whether an adjacent edge belongs to T.

Recall that $queue_i$ denotes i's message-queue, and that $contents_i(e)$ denotes e's contents of messages incoming to i ($i \in V, e \in E_i$). The discussion above suggests that we must prove that the specification [p|S|q] holds, where

$p \equiv p_1 \wedge p_2$ and $q \equiv q_1 \wedge q_2$ are defined by

$p_1 \equiv \forall i \in V\ \forall e \in E_i.\ (sn_i=\text{sleeping} \wedge se_i(e)=\text{basic})$,

$p_2 \equiv \forall i \in V\ \forall e \in E_i.\ (queue_i=\langle\rangle \wedge contents_i(e)=\langle\rangle)$,

$q_1 \equiv \forall i \in V.\ (sn_i=\text{found} \wedge (V,\{e \in E_i | se_i(e)=\text{branch}\})=T)$, and

$q_2 \equiv p_2$, with S and T as defined above.

Here [p|S|q] holds iff the following is satisfied: if execution of S is started in a state satisfying p, then S always terminates in a state satisfying q (total correctness). Consequently, the difference between {p}S{q} and [p|S|q] is that the latter specificaton implies that the program S always terminates when started in a state satisfying p.

Observe, however, that we can be more precise about the predicate q that must hold upon termination of S. Intuitively, if $e \in E_{i,j}$, and $se_i(e)=\text{branch}$ holds upon termination of S, then this implies that e is a edge of T ($i,j \in V$). Since T is an *undirected* tree, $se_j(e)=\text{branch}$ must hold then, too. Also, upon termination of S, each node should have determined, whether an adjacent edge is in T. Consequently, upon termination of S, we require that $se_i(e) \neq \text{basic}$ holds for all $e \in E_i$.

These observations lead to the specification [p|S|q'] with

$$q' \equiv q \wedge \forall i \in V\ \forall e \in E_i.\ [(se_i(e) \neq \text{basic}) \wedge \wedge \forall i,j \in V\ \forall e \in E_{i,j}.\ se_i(e)=se_j(e)],$$

where p, S, and q are as defined above.

Next, observe that S can be obtained rather easily if the network consists of one node only. Consequently, in the remainder of this paper we assume that $|V| \geq 2$ holds.

A node starts participating in the algorithm, when one of the following occurs:
- it responds to some command from a higher level procedure to initiate the algorithm, or

it receives the first (algorithm) message transmitted by some node in the graph.
A node can respond only to some command from a higher level procedure to initiate the algorithm, if it is in the sleeping state. Since the structure of such a procedure is of minor interest for the algorithm, we ignore such procedures. Instead, nodes in the graph can initiate the algorithm, according to their local information, by "awakening spontaneously". Note that many nodes can awaken spontaneously and "initiate" the algorithm. We demand, however, that a node can awaken spontaneously, only if it is in the sleeping-state.

In Gallager's algorithm, one starts with fragments of the form $(\{i\},\varnothing)$, $i\in V$. In the algorithm, each fragment finds its minimum-weight outgoing edge asynchronously with regard to other fragments. When (and if) such an edge has been found, the fragment attempts to combine with the fragment at the other end of the edge. The rules of combining have been described earlier. The part of the algorithm associated with how a fragment finds its minimum-weight outgoing edge and how to attempt combining with the fragment at the other end of that edge is called a *phase* of the fragment. In the remainder of this chapter, we consider the phase of a fragment of the form $(\{i\},\varnothing)$, and the phase of a fragment that has been formed from smaller ones at the same levels with the same minimum-weight outgoing edge.

A fragment consisting of one node only, starts its first phase when the node of that fragment awakens spontaneously, or when it receives the first algorithm-message. When a node awakens according to one of these possibilities, it determines its minimum-weight adjacent (hence, outgoing) edge (from its local table), marks this edge as a branch, and goes into the found-state (since the minimum-weight outgoing edge of its fragment has been determined). The node then sends a Connect-message with its level, i.e., 0, on the edge marked as a branch. This message serves as a request to combine with the fragment at the other end of that edge into a larger one. Sending Connect(0) by $i\in V$ also indicates the end of the first phase of a trivial fragment of the form $(\{i\},\varnothing)$ when awakening. Hereafter, it simply waits for a response from the fragment at the other end of the edge on which the Connect-message has been sent. At the first stage of the proof in [21], we have ignored the actions taken by a node, or more precisely by a fragment, when it receives such a response.

Next, we describe the actions performed by the nodes, when one (or possibly more of them) awakens spontaneously, and when two fragments are combined into a larger fragment. Node i performs its first phase when executing $S_{i,1}$ in S1 below. Node i in a fragment formed by two smaller ones at the same level with identical minimum-weight outgoing edges participates in a phase when executing $S_{i,2};S_{i,3};S_{i,4}$ in S1 below. In the program to follow, sn denotes the node-state, ln denotes the level of the fragment as far as "known" to that node, and se(e) records the status of edge e adjacent to that node. The initial values of the variables se(e) and sn are basic and sleeping, respectively; the initial values of the other variables are irrelevant. For a complete description of Gallager's algorithm the reader is referred to [7].

**program S1** (as executed by each node $i\in V$)

$S_{i,1}$

```
(1)  response to spontaneous awakening
     (can only occur at a node in the sleeping-state)
     execute procedure wake-up

(2)  procedure wake-up
     begin
        let e be the adjacent edge of minimum-weight;
        findcount:=0;se(e):=branch;ln:=0;sn:=found;
        send Connect(0) on edge e
     end

(3)  response to receipt of Connect(l) on edge e
     begin
        if sn=sleeping then execute procedure wake-up fi
```

$S_{i,2}$

```
        if ln=l
        then  if se(e)=branch
              then ln:=ln+1;fn:=w(e);sn:=find;inbranch:=e;
                    for all edges e'≠e
                    do send Initiate(ln,fn,sn) on edge e';
                       findcount:=findcount+1
                    od;
              else place received message on end of queue
              fi
        fi
     end

(4)  response to receipt of Initiate(l,f,s) on edge e
     begin
        ln:=l;fn:=f;sn:=s;inbranch:=e;
        for all edges e'≠e
        do send Initiate(ln,fn,sn) on e';
           findcount:=findcount+1
        od;
```

$S_{i,3}$

```
        best-edge:=nil;best-wt:=∞;execute procedure test
     end

(5)  procedure test
     if there are adjacent edges in state basic
     then test-edge:= minimum-weight adjacent edge in
                      state basic;
                     send Test(ln,fn) on test-edge
     else test-edge:=nil;execute procedure report
     fi

(6)  response to receipt of Test(l,f) on edge e
     begin
        if sn=sleeping then execute procedure wake-up fi;
        if ln=l
        then place received message on end of queue
        else  if fn≠f
              then send Accept on edge e
              else se(e):=rejected;
                    if test-edge≠e
                    then send Reject on e
                    else ececute procedure test
                    fi;
              fi
        fi
     end

(7)  response to receipt of Accept on edge e
     begin
        test-edge:=nil
        if w(e)<best-edge
        then best-edge:=e;best-wt:=w(e)
        else execute procedure report
        fi
     end
```

```
(8)  response to receipt of Reject on edge e
     begin se(e):=rejected;execute procedure test end

(9)  procedure report
     if findcount=0 and test-edge=nil
     then sn:=found;send Report(best-wt) on inbranch
     fi

(10) response to receipt of Report(w) on edge e
     if inbranch≠e
     then findcount:=findcount-1;
          if w<best-wt then best-wt:=w;best-wt:=e fi;
          execute procedure report
     else if sn=find
          then place received message on end of queue
          else  if w=best-wt
                then halt fi
```

$S_{i,4}$

```
                else  if w>best-wt
                      then execute procedure change-root
                      fi
                fi
          fi
     fi

(11) procedure change-root
     if se(best-wt)=branch
     then send Change-Root on best-edge
     else send Connect(ln) on best-edge;
          se(best-edge):=branch
     fi

(12) response to receipt of Change-Root
     execute procedure change-root
```

We have already given an intuitive explanation of the parts of Gallager's algorithm corresponding with the labeled parts in S1 as shown above. In [21] we have established the correctness of the program S that embodies Gallager's algorithm from properties which we derived for the program S1 above. The proof of properties for S has been structured by first concentrating on the layer $L_1 = S_{1,1} \| \ldots \| S_{n,1}$, where n is the number of nodes in the network under consideration. This layer is concerned with zero-level fragments. At this stage of the proof, we have completely ignored other communications that could affect the communication closedness of layer $L_1$. Thereafter, we have shown that a fragment F combined from two fragments F' and F" at the same level with an identical minimum-weight outgoing edge finds its minimum-weight outgoing edge, if any, and that the program terminates otherwise. To do so, we proved properties of the layers

(a) $L_2 = S_{1,2} \| \ldots \| S_{n,2}$,

(b) $L_3 = S_{1,3} \| \ldots \| S_{n,3}$.

(c) $L_4 = S_{1,4} \| \ldots \| S_{n,4}$ (when executed in states satisfying a well-chosen precondition which can be proved to be established for the "full reality" of communication) under simplifying assumptions. These assumptions are the following: in (a), we ignore all communications from nodes outside F. In (b), the assumptions in (a) have to be relaxed somewhat; otherwise verification does not make sense since in (b) nodes possibly send Test-messages to nodes outside F and could therefore receive an Accept as a response to that message. In (c) the simplifying assumptions are identical to the one in (b). *Under these simplifications, we have been able to show that the four layers mentioned above are communication closed.* Therefore, the complexity of a proof that F finds its minimum-weight outgoing edge, if any, and that the program terminates otherwise is reduced indeed. We should remark here that in case (b) above the complexity of proof can be reduced even more by applying the method of projections [14], and Martin's technique [15]. To apply Martin's technique we obtain a simplified program by replacing the tests w=best-wt in "response to receipt of Report(w) on edge e" by **false**. In case of termination of the original program, all nodes in the network would have reached a quiescent state with $\text{best-wt}_i = \infty$ for all nodes i, when executing this simplified program. From conditions satisfied in this state we are able to prove termination and a termination condition for the original program.

Also we have shown that whenever some node k receives Connect(l) and checks if $ln_k = l$ holds, then such a test is equivalent to $\neg(ln_k < l)$.

Thereafter, we have taken into account all communications that have been ignored before when reasoning in a simplified fashion about Gallager's algorithm. (At that point the possibility that low-level fragments attempt to combine with high-level fragments is incorporated in S1.) The program S as given in the appendix of [7] can then be obtained after some trivial transformations. It is interesting to note that the communication closedness of the layers as we derived earlier is destroyed when taking into account *all* communications. The intuitive reason is the following: any node i must be able to process a Connect(l) with $l < ln_i$, no matter what layer it executes. Yet, all earlier derived invariants remain valid after the addition of all possible communications since they have been chosen interference-free w.r.t. this addition, or the earlier derived properties can be easily adjusted to be valid after this addition.

From the proof we also learned that two (slight) optimizations are possible (w.r.t. the program given in [7]). The first one is already present in S1 above. If two fragments at the same level with an identical minimum-weight outgoing edge are combined into a larger one that it is not necessary that the nodes adjacent to that edge first exchange an Initiate-message as in [7]. Rather, the nodes adjacent to this edge can immediately update the relevant variables as shown in S1 above. The other optimization is the following: if a node $i \in V$ transmits a Test-message on some edge e, and it receives a Connect(l) with $l < ln_i$ on this edge before it has actually received a response on that Test-message, then there is no need to wait for this response. In this case, i will always receive a Reject-message afterwards. Consequently, it suffices for i, in this case, to continue its search for the minimum-weight outgoing edge without waiting for a response. The node j at the other end of e could then as well ignore the Test-message in such a situation, i.e., if it attempts to process a Test(l,f) with $l < ln_j$, received on an edge in state branch.

## 4. Conclusion

We have sketched the correctness of the distributed minimum-weight spanning tree algorithm of Gallager, Humblet, and Spira ([7]). We have also shown that there exist strategies to reduce the complexity of such complex correctness proofs. Basically, this reduction is achieved by introducing a certain abstraction from operational reasoning, Elrad and Francez's communication closed layers, and Martin's and Chandy & Misra's quiescence into the proof. (How the notion of quiescence can been used in the proof has not been

illustrated in this paper. For this the interested reader is referred to [21].) This allows us to reason about distributed pieces of programs under simplifying assumptions. At the final stage of the proof the assumptions implied by our abstractions must be eliminated. It then merely remains to show that properties derived during earlier stages of the proof are not invalidated (or can easily be adjusted), when taking into account communications whose occurrence we originally ignored. Moreover, it is interesting that this technique can be used to analyze other distributed programs, such as *failsafe routing algorithms* (of Merlin and Segall ([17])), *minimum path algorithms* (of Zerbib and Segall ([23])), and *maximal flow algorithms* in a network (of Segall ([18])), too.
Future work will investigate whether this technique, and the proof presented in this paper, can be extended to verify the correctness of directed minimum-weight spanning tree algorithms (see e.g., [10]). Another research topic in this field is to extend the minimum-weight spanning tree algorithm of Gallager, Humblet, and Spira to networks in the presence of failures and additions of links and nodes, i.e., to consider some failsafe version of this algorithm. We conjecture that our analysis can be also extended to the *construction* of other algorithms in this area.

Acknowledgement: we thank H. Partsch for a number of remarks that have led to a smoother presentation.

## References


[1]. Apt K.R., Francez N., de Roever W.P., A proof system for communicating sequential processes, ACM TOPLAS, 2-3, (1980).

[2] Bochmann G.V., Finite state description of communicating protocols, Computer Networks, 2,(1978).

[3] Chandy M., Misra J., An example of stepwise refinement of distributed algorithms: quiescence detection, ACM TOPLAS, 8-3, (1986).

[4] Dijkstra E., Two problems in connection with graphs, Numer. Math. 1, (1959).

[5] Elrad T., Francez N., Decomposition of distributed programs into communication-closed layers, Science of Computer Programming, 2, (1982).

[6] Even S., Graph algorithms, Computer Science Press, Inc. (USA), (1979).

[7] Gallager R.G., Humblet P.A., Spira P.M., A distributed algorithm for minimum-weight spanning trees, ACM TOPLAS, 5-1, (1983).

[8] Gerth R.T., Shrira L., On proving communication closedness of distributed layers, the Proceedings of the 6th conference on Foundations of Software Technology and Theoretical Computer Science, New Dehli, India, (1986).

[9] Hoare C.A.R., Communicating Sequential Processes, Comm. ACM, 21-8 (1978).

[10] Humblet P.A., A distributed algorithm for minimum-weight directed spanning trees, IEEE Transactions on Communications, 31-6, (1983).

[11] Kruskal J.B. , On the shortest spanning subtree of a graph and the traveling salesman problem, Proc. Am. Math. Soc., 7, (1956).

[12] Lamport L., Specifying concurrent modules, ACM TOPLAS, 5-2, (1983).

[13] Lamport L., An assertional correctness proof of a distributed algorithm, Science of Computer Programming 2-3, (1982).

[14] Lam S.S, Shankar A.U., Protocol verification via projections, IEEE Trans. on Softw. Eng., 10 4, (1984).

[15] Martin A.J., A distributed path algorithm and its correctness proof, Report Philips Research Lab. (1980, revised 1984).

[16] Misra J., Chandy K.M., Proofs of network of processes, IEEE Trans. on Soft. Eng. 7, (1981).

[17] Merlin P.M., Segall A., A failsafe distributed routing protocol, IEEE Trans. on Comm., 27-9, (1979).

[18] Segall A., Decentralized maximum flow protocols, Internal Report Technion-Israel Institute of Technology, Haifa, Israel (1980).

[19] Shankar A.U.,Lam S.S., An HDLC protocol specification and its verification using image protocols, ACM Trans. On Comp. Syst., 1-4, (1983).

[20] Stomp F.A., Methods for the analysis of protocols, manuscript (1986).

[21] Stomp F.A., de Roever W.P., A correctness proof of a distributed minimum-weight spanning tree algorithm (full paper), Internal Report 87-4, University of Nijmegen.

[22] Zwiers J., de Roever W.P., van Emde Boas P., Compositionality and concurrent networks: soundness and completeness of a proof system, Proc. 12th ICALP, LNCS 194, Springer-Verlag, New York (1985).

[23] Zerbib F.B.M., Segall A., A distributed shortest path protocol, Internal Report EE pub. no. 395, Technion-Israel Institute of Technology, Haifa, Israel (1980).

# CHAPTER 3

An extended abstract of a version of this article has been published in the *Proceedings of the 3rd International Workshop on Distributed Algorithms* (LNCS 392), Nice, France, September 26-28, 1989, pp. 242-253.

# Designing distributed algorithms
# by means of
# formal sequentially phased reasoning

F.A. Stomp* W.P. de Roever†

**Abstract:** Designers of network algorithms give elegant informal descriptions of the intuition behind their algorithms (see [GHS83, Hu83, MS79, Se82, Se83, ZS80]). Usually, these descriptions are structured as if tasks or subtasks are performed *sequentially*. From an operational point of view, however, they are performed *concurrently*. Here, we present a design principle that formally describes how to develop algorithms according to such sequentially phased explanations. The design principle is formulated using Manna and Pnueli's linear time temporal logic [MP83]. This principle, together with Chandy and Misra's technique [CM88] or Back and Sere's technique [BS89] for designing parallel algorithms, is applicable to large classes of algorithms, such as those for minimum-path, connectivity, network flow, and minimum-weight spanning trees. In particular, the distributed minimum-weight spanning tree algorithm of Gallager, Humblet, and Spira [GHS83] is structured according to our principle.

*University of Nijmegen, Department of Computer Science, Toernooiveld, 6525 ED Nijmegen, The Netherlands. E-mail address: frank@cs.kun.nl

†Eindhoven University of Technology, Department of Mathematics and Computing Science, POB 513, 5600 MB Eindhoven, The Netherlands. Email address: mcvax!eutrc3!wsinwpr

# 1 Introduction

Designers of complex network algorithms, see, e.g., [GHS83, Hu83, MS79, Se82, Se83, ZS80], usually describe their algorithms on the basis of *tasks* or *subtasks* – sometimes referred to as *phases* and *subphases*. Their (informal) descriptions are structured as if groups of nodes in the network perform these (sub)tasks *sequentially*, although in reality (i.e., operationally speaking) they are performed *concurrently*. Current design methodologies (see, e.g., [CM88, BS89]) lack an appropriate principle for *formally* developing such sequentially phased algorithms. *In this paper we formulate a formal design principle that captures this sequential structure in network algorithms.* It closely resembles the designers' intuitions as given by the informal descriptions and thus preserves the natural flavor of their original explanation. Furthermore, this principle can also be used to design formally new algorithms.

The sequential decomposition of a concurrently performed task into subtasks can already be discerned in a simple broadcast protocol, viz., Segall's PIF-protocol [Se83] (cf. also [DS80] and [F80]). Here, the whole protocol performed by the nodes in some network can be decomposed into two subtasks: the *first one* broadcasting some information and unwinding a directed tree, and the *second one* reporting that the nodes have indeed received the information. Following this pattern of sequential reasoning the distributed minimum-weight spanning tree algorithm of Gallager, Humblet, and Spira [GHS83], hereafter referred to as Gallager's algorithm, can be described in essentially four subtasks, which from a logical point of view are performed sequentially (see [SR87a, SR87b]). *That algorithm displays, however, an additional feature: that of "interference".* Expanding groups of nodes perform a certain task repeatedly, with different groups performing their tasks concurrently w.r.t. another. Now a task performed by one group can be disturbed temporarily due to interference with the task of another group. *Our design principle is geared to cope naturally with this kind of interference.*

In order to design a distributed program that solves a certain task which can be split up logically into subtasks as if they are performed sequentially, we propose the following strategy:
First develop distributed programs which solve the subtasks. Methodologies for doing so are described in [CM88] and [BS89]. Next, combine these programs to construct one which solves the whole task. Our design principle describes how to accomplish this combination. (In [CM88]

there has not been given any methodological advice how to accomplish this kind of combination. Our technique generalizes one transformation principle described in [BS89], because it is able to cope with repeatedly performed tasks and with temporary disturbances of the kind discussed above.)

In essence, it is required to prove the *verification conditions* (A) and (B) below.

(A) Prove that for each distributed program $S$, solving a subtask, the following holds: There exists a specification for $S$ consisting of, for each node $j$,

   (1) a precondition $p_j$ and a postcondition $q_j$, and

   (2) a pair of state-assertions $(I_j, T_j)$.

   $I_j$ is an invariant for the program executed by node $j$. Furthermore, $I_j$ is an invariant for program $S$; It has been incorporated in the specification in order to deal with the above-mentioned kind of interference, which occurs in, e.g., Gallager's algorithm (cf. [SR87a, SR87b]). $T_j$ expresses that node $j$ has completed its contribution to the subtask associated with program $S$.

(B) Prove that each node can participate in at most one subtask at a time and that all nodes which participate in more subtasks, participate in these subtasks in the same order.

One is then entitled to conclude that the program consisting of all (atomic) actions occurring in those programs associated with the subtasks solves the whole task as if the nodes perform the subtasks sequentially. Astonishingly, this simple design principle underlies the development of such complicated algorithms as Gallager's and those described in [Hu83, MS79, Se82, Se83, ZS80].

How can one understand the inherently sequential intuition present in this design principle for concurrent computations?

Its semantic foundation lies in considering computation sequences in a specific form in which all operations associated with one subtask are performed consecutively. Although it might not be the case at all that each computation sequence of the program solving the whole task is in this specific form, *reasoning about this program by means of computation sequences in this specific form is correct*, since any computation sequence of the program turns out to be *equivalent* to one in that form. In order to define this notion of equivalence (see [L85]) the notion of an event is needed: an event is an occurrence of the execution of some atomic action. Now each

computation sequence induces a partial ordering of its events. This partial order is a causal relation in which all events generated at a single node are ordered according to their temporal occurrence in this sequence. Additionally, in an asynchronous model of computation the event of sending a message precedes the event of receiving it; in a synchronous model these events are identical. Two computation sequences are *equivalent* if their first states are identical and if they define the same partial order of events. In essence, equivalent computation sequences differ only in the way events generated at different nodes are interleaved (w.r.t. the partial order defined by these sequences). Moreover, if two *finite* computation sequences are equivalent, then their last states coincide. This argument justifies, e.g., Elrad and Francez's *safe decomposition principle* [EF82] (cf. also [Pa88]) as demonstrated by Gerth and Shrira [GS86]. This principle states the following: if $S_{1,m} \| \cdots \| S_{n,m}$ is partially correct w.r.t. precondition $p_{m-1}$ and postcondition $p_m$ ($n \geq 1$, $m{=}1,\cdots,d$ for some natural number $d \geq 2$) and if no communication occurs between $S_{i,m}$ and $S_{j,m'}$ for $1 \leq i,j \leq n$, $i \neq j$, $1 \leq m,m' < d$, and $m \neq m'$, then $(S_{1,1};S_{1,2};\cdots;S_{1,d}) \| \cdots \| (S_{n,1};S_{n,2};\cdots;S_{n,d})$ is partially correct w.r.t. $p_0$ and $p_d$.
To reason formally about such arguments, Katz and Peled have proposed to use interleaving set temporal logic [KP87, KP88] as a formalism. Their logic allows one to reason about a program's behavior by considering only particular representatives of the program's computation sequences, such as the very sequences in the specific form introduced above.

From the discussion above it follows that if in some program, solving a certain task which can be split up logically into two subtasks as if they are performed sequentially, each node always performs operations associated with one subtask before operations associated with the other, then the following holds: each computation sequence of the program is equivalent to a computation sequence, in which all operations associated with the first subtask are performed before all operations associated with the second one. This is, e.g., the case for the program in figure 1 below, which describes the PIF-protocol [Se83] (cf. [DS80, F80]), where in order to illustrate our decomposition of a task into two subtasks in a few words, it is assumed that the network constitutes a tree.[1] The nodes perform the following task: some message $info(v)$, for a certain argument $v$, initially in the message queue of node $k$ (viewed as the root of the tree), has to be sent to all nodes in the network. Node $k$ has to be informed that all nodes in the network

[1] A decomposition is also possible in the case of an arbitrary connected network.

have received this message indeed and that the value $v$ has been recorded by them. The two subtasks constituting this task have been described above and consist of a broadcasting phase followed by a reporting phase. In the program below (see figure 1), boxes labeled $A_i^n$ indicate which operations of node $i$ are associated with the $n^{th}$ subtask ($n$=1,2). Note that the boxes do not necessarily correspond with the body of a "response" (since they are the outcome of a *semantical* analysis). *Now our principle justifies that one can reason formally about this protocol as if first $A^1$ programs are executed by the nodes, and thereafter only $A^2$ programs.* In appendix IV the specific assertions $I_j$, $T_j$, $p_j$, and $q_j$ for all nodes $j$ are defined in case of the PIF-protocol.

Our principle is a broad *semantic* generalization of Elrad and Francez's *safe decomposition principle* [EF82] (cf. also [GS86, Pa88]). Their decompositions, however, i.e., the programs (called *layers* in [EF82]) describing the subtasks, are restricted by the syntax of the whole program; This is not true for our decompositions as has already been observed above. In contrast with their principle, and the one described in [FF89], our principle also applies to reasoning about *repeatedly* performed tasks by expanded groups of nodes, such as in, e.g., Gallager's algorithm. Methods for verifying Gallager's algorithm appear in [SR87a, SR87b, CG88, WLL88]. We [SR87a, SR87b] have reasoned about its correctness on the basis of (sub)tasks. In those papers, however, the underlying proof principles have not been formulated. Neither has a formalism for them been given. Welch, Lamport, and Lynch [WLL88] give a correctness proof in the context of I/O-automata, using a (partially-ordered) hierarchy of algorithms. Chou and Gafni [CG88] consider a simplified version of Gallager's algorithm, a distributed version of Boruvka's algorithm [B26]. The problem of finding a simple proof principle for the sequentially phased reasoning of the full version of Gallager's algorithm clearly emerges in [CG88], since in the full version of that algorithm one has to cope with temporary disturbances of the kind discussed above. In order to reason about such disturbances along the lines of [CG88], another principle would be required. In our case, due to the collection of assertions $(I_j, T_j)$ for nodes $j$, merely an *interference-freedom* argument for $I_j$ and $T_j$ must be given.

The rest of this paper is organized as follows: in section 2, we introduce some notation and conventions. Our design principle is formulated in section 3. For ease of exposition we have restricted ourselves to synchronous communication. Section 4 contains some conclusions. Soundness of the design principle is proved in appendix I. In appendix II we discuss how to formulate our principle for the asynchronous case. Appendix III shows how to transform programs represented by lists of responses (cf. the program above) into our own notation for representing distributed algorithms. In Appendix IV contains a fully worked out illustration of the principle applied to the PIF-protocol.

loop executed by node $k$ (the root)

response to receipt of *info(v)* $\quad A_k^1$<br>
begin<br>
  $val_k := v$;<br>
  for all edges $e \in E_k$<br>
    do send $info(val_k)$ on edge $e$ od<br>
end

response to receipt of *ack(v)* on edge $C$ $\quad A_k^2$<br>
begin<br>
  $N_k(C) :=$ true;<br>
  if $\forall C \in E_k . N_k(C)$<br>
  then $done_k :=$true<br>
  fi<br>
end

loop executed by node $i \neq k$ (a non-root)

response to receipt of *info(v)* on edge $C$ $\quad A_i^1$<br>
begin<br>
  $val_i := v$; $inbranch_i := C$; $N_i(C) :=$true;<br>
  for all edges $e \in E_i \wedge e \neq inbranch_i$<br>
    do send $info(val_i)$ on edge $e$ od;

  if $\forall C \in E_i . N_i(C)$<br>
  then send $ack(val_i)$ on $inbranch_i$<br>
  fi<br>
end

response to receipt of *ack(v)* on edge $C$ $\quad A_i^2$<br>
begin<br>
  $N_i(C) :=$ true;<br>
  if $\forall C \in E_i . N_i(C)$<br>
  then send $ack(val_i)$ on $inbranch_i$<br>
  fi<br>
end

**Notation used:** $E_i$ denotes the set of edges adjacent to node $i$. Variable $val_i$ is used to record the argument of the info-message received by node $i$; $N_i(C)$ records whether any message has been received along edge $C$, $C \in E_i$. For node $i$ different from $k$, variable $inbranch_i$ records the identification of the edge along which the info-message has been received. (These variables are used for unwinding the directed tree.) Variable $done_k$ records whether the whole task has been completed.
Initially, node $k$'s message queue contains one info-message and the message queues of all other nodes are empty. Furthermore initially $\neg done_k$ holds for node $k$, and $\neg N_i(C)$ for all nodes $i$ and edges $C \in E_i$. The initial values of the other variables are irrelevant.

Figure 1 : Segall's PIF-protocol

## 2 Conventions and notations

A distributed algorithm is performed by nodes in a fixed, finite, and undirected network $(V, E)$, and consists of at least two nodes. The network is viewed as a graph. Two adjacent nodes communicate by means of messages. Since edges are undirected, each node can both send and receive messages along any of its adjacent edges. Except for delivering messages properly any edge can damage, lose, duplicate, and reorder messages in transit.

For ease of exposition it is assumed that communication is *synchronous*. (In appendix II we show how our results can be extended to an *asynchronous* model of communication.) In order to avoid bothering about the actual syntax of programs, distributed algorithms are represented by a triple $< V', \{p_i \mid i \in V'\}, A >$. (In appendix III we show how a program represented by lists of responses, as in e.g., section 1, can be represented by such a triple.) The interpretation of the three components is the following: $V'$ is a subset of $V$ containing all nodes that actually execute the algorithm. $\{p_i \mid i \in V'\}$ is a collection of state assertions. For all $i \in V'$, assertion $p_i$ describes the initial values of node $i$'s variables. Finally, $A$ is a collection of atomic actions which can occur when the nodes in $V'$ execute the algorithm (see the definition below). Each action $a$ has an enabling condition $en(a)$ associated with it.

Given an algorithm represented by a triple as above, it is assumed that the collection $A$ of actions can be partitioned into sets $A_j$ of node $j$'s internal actions and sets $A_{j,i}$, $i \neq j$, of actions involving a transmission of a message from node $j$ to node $i$ ($i, j \in V'$). The collection of all actions that can be performed by node $j$ (possibly simultaneously with other nodes), i.e., the set $A_j \cup \bigcup_{i \in V'} A_{j,i} \cup \bigcup_{i \in V'} A_{i,j}$, will be denoted by $act(A, j)$. For action $a \in A_j$, $en(a)$ refers to node $j$'s variables only. In this case, $en(a)$ will be denoted by $en_j(a)$. If some action $a$ involves a communication between the nodes $i$ and $j$, then $en(a)$ is the conjunction of boolean conditions $en_j(a)$ and $en_i(a)$ where for $\ell \in \{i, j\}$, $en_\ell(a)$ refers to node $\ell$'s variables only.

**Definition**

A computation sequence of an algorithm as above is a maximal sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \cdots$ such that for all $n \geq 0$ the following is satisfied: $s_n$ is some state, each $p_i$ ($i \in V'$) holds in state $s_0$, $a_n$ is an action occurring in the set $A$, action $a_n$ is enabled in state $s_n$, i.e., $en(a_n)$ holds in $s_n$, and $s_{n+1}$ is the state resulting when $a_n$ is executed in state $s_n$.

# 3 Our design principle

In this section we present a design principle that formalizes sequentially phased design of distributed algorithms. The principle itself is formulated in subsection 3.3. In subsection 3.2 correctness formulae and the *verification conditions* of the principle, i.e., conditions to be verified in order to apply the principle, are presented. Introducing the correctness formulae enables a simple and convenient formulation of our principle. Subsection 3.1 describes some basic observations for solving tasks from the class considered here.

### 3.1 General observations

Assume that a collection $V'$ of nodes performs a certain task specified by means of a pair of sets of state-assertions $\{p_i \mid i \in V'\}$ (the preconditions) and $\{q_i \mid i \in V'\}$ (the postconditions). Consequently, in order to solve this task by some distributed algorithm $\mathcal{A}$ we must find a collection of actions $A$ such that

- $\mathcal{A}$ is described by the triple $< V', \{p_i \mid i \in V'\}, A >$ and
- every finite computation sequence of $\mathcal{A}$ ends in a state for which each of the postconditions $q_i$ holds ($i \in V'$).

We shall assume that this task can be split up logically into two subtasks as if they are performed sequentially. (The general case is a straightforward extension as shown at the end of this section.) It is attractive to design $\mathcal{A}$ in two stages: In the first stage algorithms $\mathcal{B}$ and $\mathcal{C}$ are designed that solve the two subtasks. Such a decomposition enables us to concentrate on one subject at a time. Methodologies for developing these algorithms are described in [CM88] and [BS89]. In the second stage $\mathcal{A}$ itself is designed by combining algorithms $\mathcal{B}$ and $\mathcal{C}$. Our design principle describes how to accomplish this combination.

Obviously, since the whole task can be split up *logically* into two subtasks, there exist intermediate assertions $r_i$, $i \in V'$, such that the two subtasks are solved by distributed algorithms $\mathcal{B}$=$< V', \{p_i \mid i \in V'\}, B >$ and $\mathcal{C}$=$< V', \{r_i \mid i \in V'\}, C >$ (for certain sets $B$ and $C$ of actions) (cf. [CM88, BS89]). Each finite computation sequence of algorithm $\mathcal{A}$ and algorithm $\mathcal{B}$ ends in a state for which each of the assertions $r_i$ and $q_i$ respectively ($i \in V'$) holds.
The remainder of this section describes how to combine these algorithms in order to obtain $\mathcal{A}$.

### 3.2 Verification conditions

We now introduce correctness formulae and present conditions which are required for a sound application of our principle. Some conditions that algorithms $\mathcal{B}$ and $\mathcal{C}$ should satisfy in order to design $\mathcal{A}$ with this principle are described by means of correctness formulae in subsection 3.2.1. Each of them can be verified by concentrating on one algorithm at a time. Conditions referring to both $\mathcal{B}$ and $\mathcal{C}$ are formulated in subsection 3.2.2.

### 3.2.1 Correctness formulae

Let $\mathcal{D}=< V', \{pre_i \mid i \in V'\}, D >$ be an algorithm which should satisfy the following: if $\mathcal{D}$ is executed (in a state satisfying each of the preconditions $pre_i$, $i \in V'$), then every finite computation sequence ends in a state for which certain state assertions $post_i$, $i \in V'$, (the postconditions) hold. Node $j$'s computation can be characterized by means of an invariant $I_j^{\mathcal{D}}$ ($j \in V'$). Introducing such invariants is the standard technique to ensure that our design principle (see subsection 3.3) can also be used for designing algorithms in which a (sub)task performed by some group of nodes can be disturbed temporarily (due to interference of the kind discussed in section 1).
Except for the invariant $I_j^{\mathcal{D}}$, we can be more precise about node $j$'s behavior. If node $j$ has completed its participation at a certain point in some computation sequence of $\mathcal{D}$, then the postcondition $post_j$ holds and $j$ cannot perform any action from that point onwards. The states in which node $j$ cannot perform any action anymore are characterized by an assertion $T_j^{\mathcal{D}}$ ($j \in V'$).

We now introduce correctness formulae of the form
$\mathcal{D}$ **sat** $< \{I_j \mid j \in V'\}, \{T_j \mid j \in V'\}, \{post_j \mid j \in V'\} >$
for an algorithm $\mathcal{D}=< V', \{pre_i \mid i \in V'\}, D >$ and for state assertions $I_j$, $T_j$, $post_j$ ($j \in V'$). Such a formula is valid iff the following holds for every computation sequence of $\mathcal{D}$:

- For all $j \in V'$, $I_j$ holds in every state of the sequence,
- For all $j \in V'$, $T_j$ holds iff node $j$ will not execute any action in $D$ anymore, and
- For all $j \in V'$, $post_j$ holds when and if node $j$ has completed its participation in $\mathcal{D}$.

A correctness formula as above can be characterized in linear time temporal logic [MP83]. Let $\mathcal{D}=< V', \{pre_i \mid i \in V'\}, D >$. $\mathcal{D}$ **sat** $< \{I_j^{\mathcal{D}} \mid j \in V'\}, \{T_j^{\mathcal{D}} \mid j \in V'\}, \{post_j \mid j \in V'\} >$ is an abbreviation of the conjunction of the conditions (a) through (f) below. (Some of these conditions are redundant; We have included them to formalize the intuition in a natural way.) The conditions below are interpreted over all computation sequences of algorithm $\mathcal{D}$. ($\Box$ denotes the *always*-operator.)

(a) $\forall j \in V'.(pre_j \Rightarrow I_j^{\mathcal{D}})$.
Therefore, initially $I_j^{\mathcal{D}}$ holds for all nodes $j$ in $V'$.

(b) $\forall j \in V'.\Box((I_j^{\mathcal{D}} \wedge \neg T_j^{\mathcal{D}})U(I_j^{\mathcal{D}} \wedge T_j^{\mathcal{D}}))$, where $U$ denotes the weak until-operator, i.e., for all nodes $j$ in $V'$, $I_j^{\mathcal{D}}$ is an invariant and the following holds: "node $j$ participates in the algorithm until it has completed its participation".

(c) $\forall j \in V'.\forall d \in act(D,j).\Box((I_j^{\mathcal{D}} \wedge T_j^{\mathcal{D}}) \Rightarrow \neg en_j(d))$, i.e., if a certain node has completed its participation in the algorithm, then it cannot perform any action. (Cf. section 2 for the definitions of $act(D,j)$ and of $en_j(d)$.)

(d) $\forall j \in V'.\Box((I_j^{\mathcal{D}} \wedge T_j^{\mathcal{D}}) \Rightarrow \Box(I_j^{\mathcal{D}} \wedge T_j^{\mathcal{D}}))$, i.e., once a node has completed its participation in the algorithm, then it will never participate in the algorithm anymore.

(e) $\forall j \in V'.\Box((I_j^{\mathcal{D}} \wedge \neg T_j^{\mathcal{D}}) \Rightarrow \exists d \in D.(en(d)))$, i.e., if in a certain state some node has not (yet) completed its participation in algorithm $\mathcal{D}$, then the whole algorithm cannot be completed.

(f) $\forall j \in V'.\Box((I_j^{\mathcal{D}} \wedge T_j^{\mathcal{D}}) \Rightarrow post_j)$, i.e., node's $j$ postcondition $post_j$ is established when it has completed its participation in the algorithm.

### 3.2.2 Conditions for combining subtasks

Let $\mathcal{B}=< V', \{p_i \mid i \in V'\}, B >$ and $\mathcal{C}=< V', \{r_i \mid i \in V'\}, C >$ be algorithms which solve the two subtasks. Assume that

(1) $\mathcal{B}$ **sat** $< \{I_j^{\mathcal{B}} \mid j \in V'\}, \{T_j^{\mathcal{B}} \mid j \in V'\}, \{r_j \mid j \in V'\} >$ and

(2) $\mathcal{C}$ **sat** $< \{I_j^{\mathcal{C}} \mid j \in V'\}, \{T_j^{\mathcal{C}} \mid j \in V'\}, \{q_j \mid j \in V'\} >$ are satisfied.

We first impose the following condition: Each programming variable occurring in any of the assertions $p_j$, $r_j$, $q_j$ $I_j^{\mathcal{B}}$, $I_j^{\mathcal{C}}$, $T_j^{\mathcal{B}}$, and $T_j^{\mathcal{C}}$ is node $j$'s own variable. The intuition behind this restriction is that a node's precondition (or its postcondition) can be described in terms of initial (or final) values of its own variables. Also, an invariant associated with some node $j$ characterizes $j$'s computation and can therefore be expressed without any reference to variables of nodes different from $j$. Analogous, a termination condition expresses that a node has completed its participation in a certain algorithm and can be expressed in terms of its own variables.

(3) Each programming variable occurring in any of the assertions $p_j$, $r_j$, $q_j$ $I_j^{\mathcal{B}}$, $I_j^{\mathcal{C}}$, $T_j^{\mathcal{B}}$, and $T_j^{\mathcal{C}}$ is node $j$'s own variable ($j \in V'$).

In order to solve the whole task, we shall design an algorithm $\mathcal{A}$ with actions from $B$ and $C$ in which each node $j$ in $V'$ first participates in $\mathcal{B}$ and then participates in $\mathcal{C}$, provided that $j$ actually participates in both subtasks. As a consequence of this strategy, no node in $V'$ will participate in both subtasks at the same time. Therefore, we require that if a certain node has not completed its participation in one subtask, then it cannot execute any action associated with the other subtask.

Define for some assertion $P$ and for some set of actions $AC$ the predicate $disabled_\ell(P, AC)$ ($\ell \in V'$) expressing that if assertion $P$ holds, then for all actions $a$ in $AC$,$en_\ell(a)$ holds: Formally, $disabled_\ell(P, AC)$ holds iff $\Box(P \Rightarrow \forall a \in AC.\neg en_\ell(a))$ is satisfied. It is required that the following conditions are satisfied:

(4) $\forall j \in V'.disabled_j(I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{B}}, act(C, j))$ holds for all computation sequences of $\mathcal{B}$, i.e., if a certain node has not completed its participation in algorithm $\mathcal{B}$, then it cannot participate in algorithm $\mathcal{C}$, and similarly

(5) $\forall j \in V'.disabled_j(I_j^{\mathcal{C}} \wedge \neg T_j^{\mathcal{C}}, act(B, j))$ holds for all computation sequences of $\mathcal{C}$.

Also, we require that if some node has completed its participation in the second subtask, i.e., the one solved by algorithm $\mathcal{C}$, then no action associated with the first subtask which can be executed by that node is enabled. This condition ensures that every node in $V'$ that actually participates in both subtasks will participate in the first subtask before it participates in the second one.

(6) $\forall j \in V'.disabled_j(I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}}, act(B, j))$ holds for all computation sequences of $\mathcal{C}$, i.e., after completing its contribution to algorithm $\mathcal{C}$, no node can ever participate in algorithm $\mathcal{B}$. (The assertion $disabled_j$ has been defined above.)

Note that no interference-freedom of specifications has to be proved: E.g., if at some point during a computation of algorithm $\mathcal{C}$, $I_j^{\mathcal{C}} \wedge \neg T_j^{\mathcal{C}}$ holds for some node $j$, then every action $a$ associated with algorithm $\mathcal{B}$ which is performed by nodes different from $j$ does not invalidate the assertion $I_j^{\mathcal{C}} \wedge \neg T_j^{\mathcal{C}}$, because of condition (3) above.

### 3.3 The design principle

After solving the two subtasks by means of the algorithms $\mathcal{B}=< V', \{p_i \mid i \in V'\}, B >$ and $\mathcal{C}=< V', \{r_i \mid i \in V'\}, C >$ as above, formulating the design principle in order to obtain an algorithm $\mathcal{A}=< V', \{p_i \mid i \in V'\}, A >$ solving the whole task is straightforward. Observe that a node is participating in the whole task iff it is participating in one of the subtasks. Therefore, we define the set of actions $A$ as the union of the sets $B$ and $C$.

Given algorithms $\mathcal{B}$ and $\mathcal{C}$. Prove that the verification conditions (1) through (6) above are satisfied for $\mathcal{B}$ and $\mathcal{C}$. Conclude that the algorithm $\mathcal{A}= < V', \{p_i \mid i \in V'\}, \mathcal{B} \cup \mathcal{C} >$ indeed solves the whole task. More precisely, we may conclude that $\mathcal{A}$ **sat** $< \{I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}} \mid j \in V'\}, \{I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}} \mid j \in V'\}, \{q_j \mid j \in V'\} >$ holds.

Observe that as a consequence of the requirement that for any node participating in a certain subtask all the node's actions associated with the other subtask are disabled (cf. the conditions (4) and (5) above), it follows that the set of actions $\mathcal{B}$ and $\mathcal{C}$ can be chosen disjoint.

Note that we have dealt above with partial correctness only. If it is required to design an always terminating algorithm $\mathcal{A}$, then one must additionally prove a verification condition that both $\mathcal{B}$ and $\mathcal{C}$ always terminate (notation as above). This holds because the whole task terminates iff both its subtasks terminate. Formally formulating the condition that a certain algorithm terminates is straightforward and therefore omitted.

In order to establish the validity of the principle above we have shown that every finite computation sequence of $\mathcal{A}$ is equivalent (in the sense of section 1) to a finite one in which every action associated with the first subtask is performed before other actions associated with the second subtask. The proof is given in appendix I.

From the discussions above it follows that our principle can also be used for the designing algorithms hierarchically. That is, if the task solved by $\mathcal{A}$ is a subtask of yet another task, the the same principle can be applied for solving the other task.

In case the whole task can be split up into more than two subtasks we proceed as follows:
First design algorithms $\mathcal{D}$ solving the subtasks. Let the subtask solved by each $\mathcal{D}$ be described by preconditions $p_j^{\mathcal{D}}$ and postconditions $q_j^{\mathcal{D}}$ ($j \in V'$). Prove that for each such $\mathcal{D}$ there exist assertions $I_j^{\mathcal{D}}$ and $T_j^{\mathcal{D}}$ for each node $j$ in $V'$ such that $\mathcal{D}$ **sat** $\{I_j^{\mathcal{D}} \mid j \in V'\}, \{T_j^{\mathcal{D}} \mid j \in V'\}, \{q_j^{\mathcal{D}} \mid j \in V'\} >$ holds. Show that an assertion associated with some node $j$ does not depend on program variables of any node different from $j$ (cf. verification condition (3)). Then prove that each node can participate in one subtask at a time (cf. conditions (4) and (5) above). Thereafter prove that the nodes participate in the subtasks in some fixed order (cf. condition (6) above). Then conclude that the whole task is solved by an algorithm consisting of actions of all those algorithms that solve the subtasks.

# 4 Conclusion

We have presented a design principle which allows formal derivation of complex network algorithms by means of sequentially phased reasoning. This principle is applicable to a large class of algorithms (as e.g., as in [GHS83, Hu83, MS79, Se82, Se83, ZS80]) and allows structuring of their design according to logical (sub)tasks. We have decided to keep the formulation of the principle as simple as possibly. As a consequence, it is not immediately applicable for derivation of the PIF-protocol [Se83] when the network does not constitute a tree. The reason is that a message associated with the first subtask can be received by a node, when that node is participating in the second subtask (cf. section 1). In this case an adjustment of the design principle would be required. (Verification conditions (4) and (5) must be adjusted.) In essence, it has to be required that if a node is participating in the second subtask or has completed its participation in that subtask, then the arrival of a message associated with the first subtask does not affect the respective assertions attached to that node.

As structured verification and design of complex algorithms yields more insight in their correctness, we envisage that new language constructs will be designed in order to obtain better structured programs. In particular, we believe that a better structuring of programs can be achieved by means of a construct for describing subtasks and another one for building programs solving some task from programs which solve the subtasks.

In the future we will investigate how our principle can be extended for applications to network algorithms when edges and nodes can fail.

**Acknowledgement**: We thank R. Koymans and R. Gerth for valuable discussions. We also thank N. van Diepen and H. Partsch for their remarks concerning the presentation of our results.

## References


[AFR80] Apt K.R., Francez N., and de Roever W.P., A proof system for communicating sequential processes, ACM TOPLAS, 2-3 (1980).

[B26] Boruvka O., O jistém problému minimálním, Práca Moravské Přírodovědecké Společnosti (1926) (in Czech.).

[BS89] Back R.J.R. and Sere K., Stepwise refinement of action systems, Proc. of the international conference of mathematics and program construction (1989).

[CG88] Chou C.T. and Gafni E., Understanding and verifying distributed algorithms using stratified decomposition, Proc. of the ACM Symp. on Principles of Distr. Comp. (1988).

[CL85] Chandy K.M and Lamport L., Distributed snapshots: determining global states of distributed systems, ACM Trans. on Comp. Syst. 3-1 (1985).

[CM88] Chandy K.M. and Misra J., Parallel program design: a foundation, Addison-Wesley Publishing Company, Inc. (1988).

[DS80] Dijkstra E.W. and Scholten C.S., Termination detecting for diffusing computations, Information Processing Letters 1-4 (1980).

[Ev79] Even S., Graph algorithms, Computer Science Press, Inc.(USA), (1979).

[EF82] Elrad T. and Francez N., Decomposition of distributed programs into communication closed layers, Science of Computer programming, 2 (1982).

[F80] Francez N., Distributed termination, ACM-TOPLAS, 2-1 (1980).

[FF89] Fix L. and Francez N., Semantics-driven decompositions for the verification of distributed programs, manuscript (1989).

[GHS83] Gallager R.T., Humblet P.A., and Spira P.M., A distributed algorithm for minimum-weight spanning trees, ACM TOPLAS, 5-1 (1983).

[GS86] Gerth R.T. and Shrira L., On proving closedness of distributed layers, LNCS-241 (1986).

[Hu83] Humblet P.A., A distributed algorithm for minimum-weight directed spanning trees, IEEE Trans. on Comm., 31-6 (1983).

[KP87] Katz S. and Peled D., Interleaving set temporal logic, Proc. of the ACM Symp. on Principles of Distr. Comp. (1987).

[KP88] Katz S. and Peled D., An efficient verification method for parallel and distributed programs, Proc. of the REX-workshop (1988).

[L85] Lamport L., Paradigms for distributed programs: computing global states. LNCS-190 (1985).

[MP83] Manna Z. and Pnueli A., Verification of concurrent programs: A temporal proof system, Foundations of computer science IV, part 2, MC-tracts 159 (1983).

[MS79] Merlin P.M. and Segall A., A failsafe distributed routing protocol, IEEE Trans. on Comm., 27-9 (1979).

[Pa88] Pandya P.K., Compositional verification of distributed programs, Ph.D. thesis, Tata institute of fundamental research, Bombay, India (1988).

[Se82] Segall A., Decentralized maximum-flow algorithms, Networks 12 (1982).

[Se83] Segall A., Distributed network protocols, IEEE Trans. on Inf. Theory. IT29-1 (1983).

[SR87a] Stomp F.A. and de Roever W.P., A correctness proof of a distributed minimum-weight spanning tree algorithm (extended abstract), Proc. of the 7th ICDCS (1987).

[SR87b] Stomp F.A. and de Roever W.P., A fully worked out correctness proof of Gallager, Humblet, and Spira's minimum-weight spanning tree algorithm, Internal Report 87-4, University of Nijmegen (1987).

[SR88] Stomp F.A. and de Roever W.P., A formalization of sequentially phased intuition in network protocols, Internal Report 88-15, University of Nijmegen (1988).

[SS84] Schlichting R.D. and Schneider F.B., Using message passing for distributed programming, Proof rules and disciplines, ACM TOPLAS 6-3 (1984).

[WLL88] Welch J.L., Lamport L., and Lynch N.A., A lattice-structured proof of a minimum spanning tree algorithm, Proc. of the ACM Symp. on Principles of Distr. Comp. (1988).

[ZS80] Zerbib F.B.M. and Segall A., A distributed shortest path protocol, Internal Report EE-395, Technion-Israel Institute of Technology, Haifa, Israel (1980).

# Appendix I

In this appendix soundness of the design principle formulated in section 3 is proved.

In the soundness proof of the principle we use the same notation as in section 3.

Assume that the premise of the principle is satisfied. That is, assume that the conditions (1) through (6) formulated in section 3.2.2 all hold. We have to show, in order to establish the soundness of our principle, that
$\mathcal{A}$ **sat** $< \{I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}} \mid j \in V'\}, \{I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}} \mid j \in V'\}, \{q_j \mid j \in V'\} >$ holds. This amounts to proving that the conditions (a) through (f) formulated in section 3.2.1 are all satisfied for algorithm $\mathcal{A}$.

**Lemma I-1** (corresponding to condition (a) in section 3.2.1).
Under the assumption that the premise of the principle is satisfied, $\forall j \in V'.(p_j \Rightarrow (I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}}))$ holds in the first state of any computation sequence of $\mathcal{A}$.
**Proof**
This trivially follows from verification condition (1) (cf. section 3.2.2). □

Note that if some property $p$ depends on node $j$'s programming variables only, then $p$ holds in state $s$ iff $p$ holds in state $s{\downarrow}\mathrm{Var}(j)$, where $s{\downarrow}\mathrm{Var}(j)$ denotes the restriction of state $s$ to the set $\mathrm{Var}(j)$ of all node $j$'s programming variables. In the remainder of this appendix this property is referred to as property (*).

Crucial in our soundness proof is the following:

**Lemma I-2**
Suppose that the premise of the principle is satisfied. Assume that $s$ is some state in any computation sequence of algorithm $\mathcal{A}$.

(a) If, for some node $j \in V'$ and for some action $b \in act(\mathcal{B}, j)$, $en_j(b)$ holds in state $s$, then there exists a certain state $s'$ occurring in some computation sequence of algorithm $\mathcal{B}$ satisfying $s{\downarrow}\mathrm{Var}(j) = s'{\downarrow}\mathrm{Var}(j)$.

(b) If, for some node $j \in V'$ and for some action $c \in act(\mathcal{C}, j)$, $en_j(c)$ holds in state $s$, then there exists a certain state $s'$ occurring in some computation sequence of algorithm $\mathcal{C}$ satisfying $s{\downarrow}\mathrm{Var}(j) = s'{\downarrow}\mathrm{Var}(j)$.

**Proof**

Consider an arbitrary computation sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \cdots$ of algorithm $\mathcal{A}$. Let $s_x$ be some state in this sequence. We use induction of the states $s_x$, $x \geq 0$, to prove the lemma. Clearly, the lemma is true if $s_x$ is the initial state of some computation sequence of $\mathcal{A}$.

Now, assume that the lemma holds for all states $s{=}s_y$ for $0 \leq y < x$ (the induction hypothesis).

(a) If, for some node $j$ in $V'$, $en_j(b)$ holds in state $s_x$ for a certain action $b \in act(B, j)$ then either (a1) or (a2) below is true:

(a1) $\forall y < x.a_y \notin C$, i.e., in the computation sequence above state $s_x$ has been reached by executions of actions from the set $B$ only. In this case it is obvious that the lemma is satisfied.

(a2) $\exists y < x.a_y \in C$, i.e., in the computation sequence above, $s_x$ has been reached by executions of actions from $B$ and by execution of at least on action from the set $C$. Now, node $j$ cannot be involved in the execution of any action $a_z \in C$ with $z < x$. This holds because of the following:

If such an $a_z \in act(C, j')$ is the first $C$-action executed by node $j'$ in the sequence above then $I_{j'}^{\mathcal{C}} \wedge \neg T_{j'}^{\mathcal{C}}$ is satisfied in state $s_z$. (Node $j'$ has only executed $B$-actions when state $s_z$ has been reached. By the induction hypothesis, the verification conditions (1), (2), (3), and (6), and property (*) above, it follows that $I_{j'}^{\mathcal{C}} \wedge \neg T_{j'}^{\mathcal{C}}$ is satisfied in state $s_z$.) From the verification conditions (2), (3), (5), and (6), and property (*), $\forall b' \in act(B, j').\neg en_{j'}(b')$ holds in state $s_{z+1}$.

Analogous, it can be proved that if action $a_z$, $z < y$, is not the first $C$-action in the sequence above in which node $j'$ is involved, then $\forall b' \in act(B, j').\neg en_{j'}(b')$ holds in state $s_{z+1}$. We conclude that if some action $b \in act(B, j)$ is enabled in state $s_x$ then it has not performed any $C$-actions. It is now obvious that the lemma is satisfied.

(b) This case can be proved by a similar kind of reasoning as in the proof of (a) above. □

Observe that, as a consequence of property (*) and the verification conditions (1) and (2), for all states in any computation sequence of algorithm $\mathcal{A}$, $(I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}}) \wedge \neg(I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}})$ implies $(I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{B}}) \vee (I_j^{\mathcal{C}} \wedge \neg T_j^{\mathcal{C}})$, $j \in V'$. This property will be used in the following lemmata.

**Lemma I-3** (corresponding to condition (b) in section 3.2.1).

Under the assumption that the premise of the principle is satisfied,

$\forall j \in V'.\Box\big(((I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}}) \wedge \neg(I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}}))U((I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}}) \wedge (I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}}))\big)$ holds for all computation sequence of $\mathcal{A}$.

**Proof**

Consider an arbitrary computation sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \cdots$ of algorithm $\mathcal{A}$. Obviously, in order to establish the lemma it suffices to prove the following:

**Claim** :

If in a certain state $s_n$ in the sequence above action $a_n$ is executed and if $(I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}}) \wedge \neg(I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}})$ holds in state $s_n$, then $(I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}})$ holds in state $s_{n+1}$ (for all $j$ in $V'$).

**Proof** of the claim:

Assume that $(I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}}) \wedge \neg(I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}})$ holds in state $s_n$. According to the observation above, we distinguish two cases.

**Case** (i): $I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{B}}$ holds in state $s_n$.

Now, if node $j$ is involved in the execution of action $a_n$, then $a_n \in B$ holds (cf. lemma I-3 and the verification condition (4) of the principle). From lemma I-3 and the verification condition (1) it follows that $I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{B}}$ or $I_j^{\mathcal{B}} \wedge T_j^{\mathcal{B}}$ holds in state $s_{n+1}$. If, on the other hand, node $j$ is not involved in the execution of action $a_{n+1}$, then $I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{B}}$ holds in state $s_{n+1}$ (cf. verification condition (3)). We conclude that in this case the claim is satisfied.

**Case** (ii): $I_j^{\mathcal{C}} \wedge \neg T_j^{\mathcal{C}}$ holds in state $s_n$.

If node $j$ is involved in the execution of action $a_n$, then $a_n \in C$ (cf. lemma I-3 and the verification condition (5)) and, either $I_j^{\mathcal{C}} \wedge \neg T_j^{\mathcal{C}}$ or $I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}}$ hold in state $s_{n+1}$ (cf. verification condition (2)). The claim then follows from the fact that $I_j^{\mathcal{C}} \wedge \neg T_j^{\mathcal{C}}$ implies $(I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}})$ and the fact that $I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}}$ implies $(I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}})$.

If node $j$ is not involved in the execution of action $a_n$, then the claim follows from the verification condition (3). $\Box$

**Lemma I-4** (corresponding to condition (c) in section 3.2.1).

Under the assumption that the premise of the principle is satisfied,

$\forall j \in V'.\forall a \in act(B \cup C, j).\Box\big(((I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}}) \wedge (I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}})) \Rightarrow \neg en_j(a)\big)$ holds for all computation sequences of $\mathcal{A}$.

**Proof**

Assume that at some point in a computation sequence of $\mathcal{A}$, $(I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}}) \wedge (I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}})$ holds. Then $(I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}})$ holds, too. If at that point in the sequence for all nodes $j' \in V'$ and for all actions $a$ from the set $act(B \cup C, j')$, $\neg en_{j'}(a)$, then we are done.

Otherwise, i.e., $\exists j' \in V'.\exists a \in act(B \cup C, j').en_{j'}(a)$ holds. In this case, for all $a \in act(B \cup C, j)$, $\neg en_j(a)$ is satisfied as a consequence of lemma I-3, property (*), and the verification conditions (2) and (5). $\square$

**Lemma I-5** (corresponding to condition (d) in section 3.2.1).

Under the assumption that the premise of the principle is satisfied,

$\forall j \in V'.\square\big(((I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}}) \wedge (I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}})) \Rightarrow \square((I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}}) \wedge (I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}}))\big)$ holds for all computation sequence of $\mathcal{A}$.

**Proof**

Assume that in some state during a computation of $\mathcal{A}$, $(I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}}) \wedge (I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}})$ holds. Then $(I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}})$ holds, too. Node $j$ cannot execute any action in such a state (cf. lemma I-3). The assertion $(I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}})$ is preserved under all actions from the set $B \cup C$ which can be performed by nodes different from $j$, cf. the verification condition (3). The lemma is, obviously, satisfied.
$\square$

As a preparation for the proof that condition (e), formulated in section 3.2.1, holds for algorithm $\mathcal{A}$, we first have the following lemma, concerning equivalent computation sequences of a certain algorithm. (This notion of equivalence has been introduced in section 1.)

**Lemma I-6**

Suppose that $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \cdots s_x \xrightarrow{a_x} s_{x+1} \xrightarrow{a_{x+1}} s_{x+2} \xrightarrow{a_{x+2}} s_{x+3} \ldots$ is a computation sequence of some algorithm. Assume that the executions of the actions $a_x$ and $a_{x+1}$ involve distinct nodes. Then there exists some state $s'_{x+1}$, such that

$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \cdots s_x \xrightarrow{a_x} s_{x+1} \xrightarrow{a_{x+1}} s_{x+2} \xrightarrow{a_{x+2}} s_{x+3} \ldots$ and

$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \cdots s_x \xrightarrow{a_{x+1}} s'_{x+1} \xrightarrow{a_x} s_{x+2} \xrightarrow{a_{x+2}} s_{x+3} \ldots$

are equivalent computation sequences of $\mathcal{A}$.

**Proof**

Let $s'_{x+1}$ be the state resulting from execution of action $a_{x+1}$ in state $s_x$. Note that this action does not affect variables of nodes different from the ones involved in the execution of that action. From the assumption that the execution of the actions $a_x$ and $a_{x+1}$ involve distinct nodes, it then, obviously, follows that $s_{x+2}$ is the state resulting when action $a_x$ is executed in state $s'_{x+1}$. $\Box$

As a consequence of this lemma and of the proof of lemma I-2, we have:

**Lemma I-7**

Suppose that $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \cdots s_x \xrightarrow{a_x} s_{x+1} \xrightarrow{a_{x+1}} s_{x+2} \xrightarrow{a_{x+2}} s_{x+3} \cdots$ is a finite computation sequence of algorithm $\mathcal{A}$. Assume that the premise of the principle is satisfied. Furthermore, assume that that $a_x \in C$ and $a_{x+1} \in B$ hold. Then there exists some state $s'_{x+1}$, such that $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \cdots s_x \xrightarrow{a_x} s_{x+1} \xrightarrow{a_{x+1}} s_{x+2} \xrightarrow{a_{x+2}} s_{x+3} \cdots$ and
$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \cdots s_x \xrightarrow{a_{x+1}} s'_{x+1} \xrightarrow{a_x} s_{x+2} \xrightarrow{a_{x+2}} s_{x+3} \cdots$
are equivalent computation sequences of $\mathcal{A}$. $\Box$

**Lemma I-8** (corresponding to condition (e) in section 3.2.1).

Under the assumption that the premise of the principle is satisfied,
$\forall j \in V'.\Box\big(((I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}}) \wedge \neg(I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}})) \Rightarrow \exists a \in B \cup C.(en(a))\big)$ holds for all computation sequence of $\mathcal{A}$.

**Proof**

Consider an arbitrary computation sequence $seq \equiv s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ of algorithm $\mathcal{A}$. Assume, in order to obtain a contradiction, that in a certain state $s_n$ of this sequence for some node $j \in V'$, $((I_j^{\mathcal{C}} \vee I_j^{\mathcal{C}}) \wedge (I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{C}}) \wedge \forall a \in act(B \cup C).\neg en(a))$ holds. Then this state is a final state in the sequence. Hence, the sequence is finite. We now repeatedly apply lemma I-7 in order to obtain an equivalent computation sequence of $\mathcal{A}$ in which all $B$-actions are performed before all $C$-actions. Let $seq' \equiv s_0 \xrightarrow{a'_0} s'_1 \xrightarrow{a'_1} \cdots s'_x \xrightarrow{a'_x} \cdots s_n$ be the resulting sequence, where action $a_x$ is the first $C$-action taken in this sequence. (Observe that the sequence $seq'$ ends in state $s_n$.) In state $s_x$, for all $j \in V'$, $I_j^{\mathcal{B}} \wedge T_j^{\mathcal{B}}$ holds. (Otherwise, for some $j \in V'$, $I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{B}}$ is satisfied, which implies that at least one $B$-action is enabled in state $s_x$, cf. verification condition (1). Each node which is involved in this action cannot perform any action from $C$, cf. verification condition (4). This implies, however, that the sequence $seq'$ is not maximal; Contradiction.) It follows that the sequence $s'_x \xrightarrow{a'_x} \cdots s_n$ is a computation sequence of algorithm $\mathcal{B}$. From verification condition (2), we obtain that $I_{j'}^{\mathcal{C}} \wedge T_{j'}^{\mathcal{C}}$ holds, for all $j' \in V'$, in state $s_n$. This contradicts the assumption that $I_j^{\mathcal{C}} \wedge \neg T_j^{\mathcal{C}}$ holds in this state.

**Lemma I-9** (corresponding to condition (f) in section 3.2.1).

Under the assumption that the premise of the principle is satisfied,

$\forall j \in V'.\Box\big(((I_j^{\mathcal{B}} \vee I_j^{\mathcal{C}}) \wedge (I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}})) \Rightarrow q_j\big)$ holds for all computation sequence of $\mathcal{A}$.

**Proof**

This is a consequence of property (*), lemma I-3, and the verification conditions (2) and (3). □

The soundness of the principle now follows from the lemmata I-1, I-3, I-4, I-5, I-8, and I-9 above.

# Appendix II

The design principle formulated in section 3.3 can straightforwardly be extended to an asynchronous model of computation. This is shown below. For ease of exposition we assume, for this appendix, that communication is perfect.

Assume that communication is *asynchronous*. In order to design an algorithm which solves a certain task, described by preconditions $p_i$ and postconditions $q_i$ ($i \in V'$), we follow the same strategy as before:

(1) Find intermediate assertions $r_i$ such that the two subtasks can be described by the collection of preconditions $p_i$, respectively $r_i$, and postconditions $r_i$, respectively $q_i$, for $i$ in $V'$ (cf. section 3.1).

(2) Design algorithms $\mathcal{B} = < V', \{p_i \mid i \in V'\}, B >$ and $\mathcal{C} = < V', \{r_i \mid i \in V'\}, C >$ which solve the two subtasks (cf. [CM88, BS89]).[2]

(3) Prove that the verification conditions (1) through (6) below are all satisfied.

(4) Conclude that the algorithm $< V', \{p_i \mid i \in V'\}, A \cup B >$ solves the whole task (cf. section 3.3).

The verification conditions of the design principle are essentially the same as those formulated in section 3. Now, however, we have to incorporate that fact that communication is asynchronous. In order to formulate formally these verification conditions we use, as in [SS84], the *auxiliary proof variables* $\sigma_j(e)$ and $\rho_j(e)$ ($j \in V'$, $e \in E_j$). They are used to reason about communication. $\sigma_j(e)$ records the sequence of messages transmitted by node $j$ along edge $e$; $\rho_j(e)$ records the sequence of messages received by node $j$ along edge $e$. For nodes $j, k$ and edges $e \in E_j \cap E_k$, the property $\rho_j(e) \leq \sigma_k(e)$ is preserved by any action, see [SS84]. I.e., if edge $e$ connects the nodes $j$ and $k$, then the sequence of all messages received by node $j$ along $e$ is a prefix of the sequence of all messages transmitted by node $k$ along $e$. These variables are changed when a node transmits or receives a message; They are not changed during execution of an internal action.

(1) Find assertions $I_j^{\mathcal{B}}$ and $T_j^{\mathcal{B}}$, for $j$ in $V'$ and

(2) Find assertions $I_j^{\mathcal{C}}$ and $T_j^{\mathcal{C}}$, for $j$ in $V'$, having the same interpretation as in section 3.

[2] It is assumed that the set of all atomic actions for each node $j$ can be partitioned into a set of $j$'s internal actions, a set of $j$'s actions which involve the transmission of a message, and a set of $j$'s actions involving the receipt of some message.

Of course, we have to reformulate the correctness formulae (see section 3) now incorporating an *asynchronous* model of computation. Let $\mathcal{D}=< V', \{pre_i^{\mathcal{D}} \mid i \in V'\}, D >$ be some algorithm.
$\mathcal{D}$ **sat** $< \{I_j^{\mathcal{D}} \mid j \in V'\}, \{T_j^{\mathcal{D}} \mid j \in V'\}, \{post_j \mid j \in V'\} >$ holds iff each of the following conditions (a) through (f) is satisfied:

(a) $\forall j \in V'.(pre_j^{\mathcal{D}} \Rightarrow I_j^{\mathcal{D}}) \wedge$
$\wedge\, \forall j, k \in V'.\forall e \in E_j \cap E_k.(pre_j^{\mathcal{D}} \Rightarrow \rho_j(e) \leq \sigma_k(e))$ holds for all computation sequences of $\mathcal{D}$.
Thus, initially the assertion $I_j^{\mathcal{D}}$ holds. In addition, the sequence of all messages received by any node along a certain edge is a prefix of the sequence of all messages transmitted by the node at the other end of that edge holds initially. (From the discussion above, it follows that the latter property is an invariant for algorithm $\mathcal{D}$.)

(b) This condition is the same as condition (b) formulated in section 3.2.1.

Let $Int_j^{\mathcal{D}} \subseteq D$ denote the set of node $j$'s internal actions, let $Rec_j^{\mathcal{D}}(e) \subseteq D$ denote the set of node $j$'s actions which involve the receipt of a message along edge $e$, and let $Sen_j^{\mathcal{D}}(e) \subseteq D$ denote the set of node $j$'s actions which involve the transmission of a message along edge $e$ ($j \in V', e \in E_j$). Hereafter, $IS_j^{\mathcal{D}}$ will denote the set $Int_j^{\mathcal{D}} \cup \bigcup_{e \in E_j} Sen_j^{\mathcal{D}}(e)$.

(c) $\forall j \in V'.\forall d \in IS_j^{\mathcal{D}}.\Box((I_j^{\mathcal{D}} \wedge T_j^{\mathcal{D}}) \Rightarrow \neg en_j(d)) \wedge$
$\wedge\, \forall j, k \in V'.\forall e \in E_j \cap E_k.\Box((I_j^{\mathcal{D}} \wedge T_j^{\mathcal{D}}) \Rightarrow \rho_j(e) = \sigma_k(e))$ holds for all computation sequences of $\mathcal{D}$,
i.e., if a certain node has completed its participation in the algorithm, then it cannot perform any internal action or any action which involves the transmission of a message (the first conjunct), and it cannot receive any message (the second conjunct).

(d) This condition is the same as condition (d) formulated in section 3.2.1.

(e) $\forall j \in V'.\Box((I_j^{\mathcal{D}} \wedge \neg T_j^{\mathcal{D}}) \Rightarrow (\exists k \in V'.\exists d \in IS_k^{\mathcal{D}}.en(d)) \vee$
$\vee(\exists k, m \in V'.\exists e \in E_k \cap E_m.\rho_k(e) < \sigma_m(e)))$ holds for all computation sequences of $\mathcal{D}$. Here, for sequences $t$ and $u$, $t < u$ denotes that $t$ is a proper prefix of $u$.
This condition expresses the following: if a certain node has not yet completed its participation in the algorithm, then at least one node can perform some internal action or some action which involves the transmission of a message, or at least one node has transmitted a message along one of its adjacent channels and this message has not yet been received by the node at the other end of that edge.

(f) This condition is the same as condition (f) formulated in section 3.2.1.

Then we reformulate the conditions (3) through (6) from section 3.2.2 for an asynchronous model of computation.

(3) Each programming variable occurring in any of the assertions $p_j$, $r_j$, $q_j$, $I_j^{\mathcal{B}}$, $I_j^{\mathcal{C}}$, $T_j^{\mathcal{B}}$, $T_j^{\mathcal{C}}$ is node $j$'s own variable. In addition, if some proof variable $\rho_\ell(e)$ or $\sigma_\ell(e)$ occurs in any of these assertions, then $\ell=j$ and $e \in E_j$ hold.

(4) $\forall j \in V'.disabled(I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{B}}, IS_j^{\mathcal{C}}) \wedge$
$\wedge\, \forall j, k \in V'.\forall e \in E_j \cap E_k.disabled(I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{B}}, Sen_k^{\mathcal{C}}(e))$ holds for all computation sequences of $\mathcal{B}$. Here, for assertions $P$ and sets of actions $AC$, $disabled(P, AC)$ holds iff in any state satisfying $P$ all actions in the set $AC$ are disabled. (Its formal definition is straightforward and therefore omitted.) Consequently, this condition expresses the following: if a certain node has

not completed its participation in algorithm $\mathcal{B}$, then it can neither perform an internal action nor a send-action occurring in $\mathcal{C}$ (the first conjunct), and it cannot receive a message associated with the second subtask (the second conjunct). The latter holds, because if the node is participating in the first subtask, then none of its neighbors can send such messages.

Analogously we have

(5) $\forall j \in V'. disabled(I_j^{\mathcal{C}} \wedge \neg T_j^{\mathcal{C}}, IS_j^{\mathcal{B}}) \wedge$
$\wedge\ \forall j, k \in V'. \forall e \in E_j \cap E_k. disabled(I_j^{\mathcal{C}} \wedge \neg T_j^{\mathcal{C}}, Sen_k^{\mathcal{B}}(e))$ holds for all computation sequences of $\mathcal{C}$.

In order to ensure that a certain node can participate in the second subtask only after completing the first subtask (see condition (6), section 3.2.2), we impose the following condition:

(5) $\forall j \in V'. disabled(I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}}, IS_j^{\mathcal{B}}) \wedge$
$\wedge\ \forall j, k \in V'. \forall e \in E_j \cap E_k. disabled(I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}}, Sen_k^{\mathcal{B}}(e))$ holds for all computation sequences of $\mathcal{C}$.

**Remark**: We have decided to keep the formulation of the design principle above as simple as possible. As a consequence we have required that no node $k$ can send some message to another node $j$ which is participating in a different subtask than $k$ (cf. conditions (4) and (5) above). Although the above principle is applicable to a large class of algorithms, one could have been less restrictive: if some node $j$ is participating in some subtask and $j$ has some message in its queue associated with another subtask (such a situation can be recognized by tagging messages), then processing this message is delayed until $j$ is participating, or starts to participate, in the subtask associated with that message.

# Appendix III

We claim that any distributed program can be represented by a triple of the kind introduced in the paper. The validity of this claim is illustrated below by showing that any program described by a list of responses as in section 1, and as in [GHS83], can be represented by such a triple. As an example we show how the program of section 1 can be represented by such a triple.
In order to keep the presentation reasonably short we assume that communication is asynchronous and perfect.

Let $S$ be a program described by a list of responses. Our objective is to represent $S$ by a triple $\mathcal{A}$ of the kind mentioned above, such that for any computation sequence *seq* starting in an initial state satisfying some predescribed precondition the following holds: *seq* is a computation sequence of $S$ iff *seq* is a computation sequence of $\mathcal{A}$. It is obvious that the only difficulty in defining $\mathcal{A}$ is the definition of its set of atomic actions. In order to define this set we first assign labels to control points in $S$. (Such a control point is an entry- or exit-point of some atomic action occurring in $S$.) Then we introduce for each node $j$ a fresh variable $loc_j$. This variable is used to simulate node $j$'s program counter when $S$ is executed. Each (atomic) action $a$ which can be performed by $j$ is then represented by the atomic action $a;loc_j{:=}l_1$ where $l_1$ denotes the label assigned to $a$'s exit-point. The enabling condition of the action $a;loc_j{:=}l_1$, i.e., $en(a;loc_j{:=}l_1)$, is given by $loc_j = l_2$ where $l_2$ denotes the the label assigned to $a$'s entry-point. Except for these actions, we also define for each node $j$ two kinds of other actions: The first one corresponds to actions removing messages from adjacent edges and placing these received messages at the end of node $j$'s message queue. These kind of actions do not refer to the variable $loc_j$ and can occur at any time in every computation sequence, provided that some message has arrived at node $j$ (cf. (1) and (2) below). The second kind of actions corresponds to removing the first element from node $j$'s message queue, provided that it is non-empty, and setting the variable $loc_j$ to the label assigned to the "first" entry point of the respective response.
It is important that we have made explicit two tacit assumptions which are quite common when distributed algorithms are described by means of lists of responses:

(1) A message that has arrived at a node along one of its adjacent edges can be removed afterwards from that channel at any point in the computation.

(2) After the receipt of a message a node can resume its execution at the point where the node has been interrupted by the arrival of that message.

**Example:**

When labels have been assigned to the control points of the program of section 1 we obtain:

`loop executed by node` $k$

$A_k^1$:

`response to receipt of` *info(v)*<br>
`begin`<br>
$l_{k,1}$: $val_k := v$;<br>
$l_{k,2}$: `for all edges` $e \in E_k$<br>
`do send` $info(val_k)$ `on edge` $e$ `od`<br>
$l_{k,3}$:<br>
`end`

$A_k^2$:

`response to receipt of` *ack(v)* `on edge C`<br>
`begin`<br>
$l_{k,C,4}$: $N_k(C)$:= `true`;<br>
$l_{k,C,5}$: `if` $\forall C \in E_k.N_k(C)$<br>
`then` $l_{k,C,6}$: $done_k$:= `true`<br>
`fi`<br>
$l_{k,C,7}$:<br>
`end`

`loop executed by node` $i \neq k$

$A_i^1$:

`response to receipt of` *info(v)* `on edge C`<br>
`begin`<br>
$l_{i,C,1}$: $val_i := v$;<br>
$l_{i,C,2}$: $inbranch_i := C$;<br>
$l_{i,C,3}$: $N_i(C)$:= `true`;<br>
$l_{i,C,4}$: `for all edges` $e \in E_i \wedge e \neq inbranch_i$<br>
`do send` $info(val_i)$ `on edge` $e$ `od`;<br>
$l_{i,C,5}$: `if` $\forall C \in E_i.N_i(C)$<br>
`then` $l_{i,C,6}$: `send` $ack(val_i)$ `on` $inbranch_i$<br>
`fi`<br>
$l_{i,C,7}$:<br>
`end`

$A_i^2$:

`response to receipt of` *ack(v)* `on edge C`<br>
`begin`<br>
$l_{i,C,8}$: $N_i(C)$:= `true`;<br>
$l_{i,C,9}$: `if` $\forall C \in E_i.N_i(C)$<br>
`then` $l_{i,C,10}$: `send` $ack(val_i)$ `on` $inbranch_i$<br>
`fi`<br>
$l_{i,C,11}$:<br>
`end`

Figure 2: Segall's PIF-protocol after assigning labels to control points.

From now on, subscripts $i$ and $k$ are omitted when they are clear from the context.

Below, except for the labels of control points assigned explicitly, we have additionally introduced a label *at-queue*. Intuitively, node $i$ is at the control point labeled *at-queue*, when it tests whether its message queue is non-empty. Node $i$ evaluates the boolean expression $queue \neq <>$ for testing whether its queue is not empty. If it is non-empty, then the type and the channel identification of the first element are determined by evaluating *type*(first(queue)) and *chan*(first(queue)), respectively. Thereafter the argument of the first message in the queue is determined by evaluating *arg*(first(queue)) and is recorded. Then the first element is removed from the queue by executing the assignment queue:=rest(queue), and node's $i$ variable $loc_i$ is set to the entry point of the respective response.

Appending an element $M$ to the end of some queue $q$ will be denoted by $q$:=$q$ ^ M.

We next show node $k$'s actions when the program above is represented as a triple. (Below, C ranges over node $k$'s adjacent edges.)

$a_{k,1}$: v:=arg(first(queue)); queue:=rest(queue); loc:= $l_{k,1}$,

*en*( $a_{k,1}$): loc=*at-queue*$\wedge$ type(first(queue))=info.

$a_{k,2}$: val:=v; loc:= $l_{k,2}$

*en*( $a_{k,2}$): loc= $l_{k,1}$.

$a_{k,3}$: **for all edges** e $\in E_k$ **do send** info(val) **on edge** e **od**; loc:=*at-queue*,

*en*( $a_{k,3}$): loc= $l_{k,2}$.

$a_{k,C,4}$: v:=arg(first(queue)); queue:=rest(queue); loc:= $l_{k,C,4}$,

*en*( $a_{k,C,4}$): loc=*at-queue*$\wedge$ type(first(queue))=ack $\wedge$ chan(first(queue))=C.

$a_{k,C,5}$: N(C):=**true**; loc:= $l_{k,C,5}$,

*en*( $a_{k,C,5}$): loc= $l_{k,C,4}$.

$a_{k,C,6}$: **if** $\forall C \in E_k.N(C)$ **then** loc:= $l_{k,C,6}$ **else** loc:=*at-queue* **fi**,

*en*( $a_{k,C,6}$): loc= $l_{k,C,5}$.

$a_{k,C,7}$: done:=**true**; loc:=*at-queue*,

*en*( $a_{k,C,7}$): loc= $l_{k,C,6}$.

and finally $a_{k,C,8}$: **receive** msg **on edge** C; $a_{k,C,8}$: queue:=queue ^ (msg,C).

$a_{k,1}$, $a_{k,2}$, $a_{k,3}$ are those actions of node $k$ associated with the first subtask (cf. section 1). The other actions of node $k$ shown above are all associated with the second subtask.

The actions which can be performed by nodes different from $k$ can be determined analogously and are therefore omitted.

# Appendix IV

Below we show how our decomposition principle of appendix II can be applied to obtain the program of section 1. In particular, the invariants $I_j^B$, $I_j^C$ and the termination conditions $T_j^B$, $T_j^C$, for $j$ in $V$, are defined explicitly for this example.

Communication is assumed to be asynchronous and perfect.

It is assumed that some designer has already solved both subtasks discerned in the PIF-protocol (see section 1). Consequently, it now suffices to define the invariants and termination conditions in order to combine these programs. As a preparation for this we first have the following definition:

**Definition**

Let node $k$, the initiator of the protocol, be given.

(a) Let for $i \in V$, $dist(i,k)$ denote the distance between node $i$ and node $k$, i.e., $dist(i,k)$ denotes the minimum number of edges on any path between the nodes $i$ and $k$.

(b) For all $i \in V$, $C \in E_i$, $D_i(C)$ denotes the distance from node $k$ to the node different from node $i$ that is adjacent to edge C. Thus, $D_i(C) = n$ holds iff there exists some $j \neq i$ such that $C \in E_i \cap E_j$ and $dist(j,k)=n$ are satisfied (for nodes $i \in V$ and natural numbers $n$).
□

In the proofs of the verification conditions of our transformation principle, the following properties are used:

**Lemma**

(a) For all $C \in E_k$, $D_k(C)$=1 holds.

(b) For all nodes $i \in V$ and for all edges $C \in E_i$ the following holds: if $dist(i,k)=n$, then $D_i(C) = n\text{-}1 \vee D_i(C) = n+1$ is satisfied.

(c) For all nodes $i, j \in V$ and for edges $C \in E_i \cap E_j$, if $dist(i,k)=n$ and $dist(j,k)=n+1$, then $D_i(C) = n+1$ and $D_j(C) = n$ holds.

(d) If the graph $(V,E)$ constitutes a tree, then for all nodes $i \neq k$, $i \in V$, there exists exactly one edge $C \in E_i$ satisfying $D_i(C) = dist(i,k)\text{-}1$. □

The proof of the lemma above follows from elementary properties from graph-theory [Ev79] and is therefore omitted.

As has been argued in section 3, it is attractive to design a program describing the PIF-protocol in two stages. In the first stage the program solving the first subtask, could have been described by the program $B$ consisting of those actions associated with the programs $A_i^1$ of section 1 (cf. also appendix III). In the second stage program $C$, consisting of all actions associated with the programs $A_i^2$ of section 1 solving the second subtask, could have been developed.

Below, in the definitions of the respective assertions, we have used the auxiliary proof variables $\sigma_i(C)$ and $\rho_i(C)$ ($i \in V$ and $C \in V$). These kinds of variables have been discussed in appendix II. Recall that $\sigma_i(C)$ records the sequence of messages sent by node $i$ along edge $C$ and that $\rho_i(C)$ records the sequence of messages received by node $i$ along edge $C$.
In the sequel $\mid q \mid$ denotes the length of queue $q$, i.e., $\mid q \mid$ denotes the number of elements in $q$; For queues $q$, $q[n]$ denotes the $n^{th}$ element in $q$ ($1 \leq n \leq\mid q \mid$).

The initial states of algorithm $B$ are described by the assertions $p_j$, $j \in V$, defined below.

For node $k$, $p_k$ is defined as the conjunction of

- $loc_k$=*at-queue*$_k$ (cf. the discussion in appendix III),
- $queue_k$=$< info(w) >$ (node $k$'s queue contains only the message $info(w)$),
- $\neg done_k$ (node $k$ has not been informed that the other nodes have received the *info*-messages),
- $\forall C \in E_k.\neg N_k(C)$ (node $k$ has not recorded that it has received a message along any of its adjacent edges),
- $\forall C \in E_k.( \rho_k(C)=<> \wedge \sigma_k(C)=<>)$ (node $k$ has neither sent and nor received messages along any of its adjacent edges), and
- Tree$(V, E) \wedge \mid V \mid \geq 2$ (the graph $(V,E)$ constitutes a tree and $V$ consists of at least two nodes).

For nodes $j$ different from $k$, $p_j$ is defined as the conjunction of

- $loc_j$=*at-queue*$_j$ (cf. the discussion above),
- $queue_j$=$<>$ (node $j$'s queue is empty),

- $\forall C \in E_j.\neg N_j(C)$ (node $j$ has not recorded that it has received a message along any of its adjacent edges),

  $\forall C \in E_j.(\ \rho_j(C)=<> \wedge \sigma_j(C)=<>)$ (node $j$ has neither sent and nor received messages along any of its adjacent edges), and
- $\text{Tree}(V, E) \wedge \mid V \mid \geq 2$ (see above).

The final states of algorithm $\mathcal{B}$ are characterized by assertions $q_j$: $q_j \equiv I_j^{\mathcal{B}} \wedge T_j^{\mathcal{B}}$ holds where $I_j^{\mathcal{B}}$ and $T_j^{\mathcal{B}}$ ($j \in V$) are defined below.

For node $k$, the assertion $I_k^{\mathcal{B}}$ is defined as the conjunction of

  $\text{Tree}(V, E) \wedge \mid V \mid \geq 2$,
- $\forall C \in E_k.(\ \rho_k(C)=<>)$,
- $\forall C \in E_k.\ \neg N_k(C)$,
- $\neg done_k$, and
- the disjunction of
  - $\big(loc_k{=}at\text{-}queue_k \wedge queue_k{=}< info(w) > \wedge \forall C \in E_k.(\ \sigma_k(C)=<>)\big)$ (satisfied initially),
  - $\big(loc_k{=}l_{k,1} \wedge queue_k{=}<> \wedge \forall C \in E_k.(\ \sigma_k(C)=<>) \wedge v_k = w\big)$ (satisfied after node $k$ has removed the *info*-message from its queue),
  - $\big(loc_k{=}l_{k,2} \wedge queue_k{=}<> \wedge \forall C \in E_k.(\ \sigma_k(C)=<>) \wedge val_k = w\big)$ (satisfied after node $k$ has recorded the argument of the *info*-message), and
  - $\big(loc_k{=}at\text{-}queue_k \wedge queue_k{=}<> \wedge \forall C \in E_k.(\ \sigma_k(C)=< info(w) >) \wedge val_k = w\big)$ (satisfied after node $k$ has broadcasted the *info*-message).

The assertion $T_k^{\mathcal{B}}$ is defined to express that node $k$ has broadcasted the info-message. Formally, we define $T_k^{\mathcal{B}} \equiv \forall C \in E_k.\ \sigma_k(C)=< info(w) >$).

For nodes $j$ different from node $k$, $I_j^{\mathcal{B}}$ is defined as the conjunction of
- $\text{Tree}(V, E) \wedge \mid V \mid \geq 2$,
- $\forall C \in E_j.\big((D_j(C) = dist(j,k)\text{-}1 \Rightarrow \sigma_j(C)=<>) \wedge$
  $\wedge(D_j(C) = dist(j,k)+1 \Rightarrow \rho_j(C)=<>)\big)$,

  i.e., if the graph $(V, E)$ is considered to be rooted at node $k$, then $j$ does not send any message uptree and it does not receive messages from nodes downtree,

- $\forall C \in E_j.(D_j(C) = dist(j,k)+1 \Rightarrow \neg N_j(C))$,

  i.e., if the graph $(V, E)$ is considered to be rooted at node $k$, then node $j$ cannot record that a message has been received from nodes downtree, and

- the disjunction of

  - $\big(loc_j = at\text{-}queue_j \wedge queue_j = <> \wedge$
    $\wedge\ \forall C \in E_j.\neg N_j(C) \wedge \forall C \in E_j.(\ \rho_j(C) = <> \wedge\ \ \sigma_j(C) = <>)\big)$
    (satisfied initially),

  - $\big(loc_j = at\text{-}queue_j \wedge$
    $\wedge\ \exists C \in E_j.(queue_j = < info(w), C > \wedge\ D_j(C) = dist(j,k)\text{-}1 \wedge\ \ \rho_j(C) = < info(w) >) \wedge$
    $\wedge\ \forall C \in E_j.\neg N_j(C) \wedge \forall C \in E_j.(D_j(C) = dist(j,k)+1 \Rightarrow \sigma_j(C) = <>)\big)$
    (satisfied after node $j$ has received the *info*-message),

  - $\big(\exists C \in E_j.(loc_j = l_{j,C,1} \wedge D_j(C) = dist(j,k)\text{-}1 \wedge\ \ \rho_j(C) = < info(w) >) \wedge$
    $\wedge\ \forall C \in E_j.\neg N_j(C) \wedge \forall C \in E_j.(D_j(C) = dist(j,k)+1 \Rightarrow \sigma_j(C) = <>) \wedge$
    $\wedge\ queue_j = <> \wedge\ v_j = w\big)$
    (satisfied after node $j$ has removed the *info*-message from its queue),

  - $\big(\exists C \in E_j.(loc_j = l_{j,C,2} \wedge D_j(C) = dist(j,k)\text{-}1 \wedge\ \ \rho_j(C) = < info(w) >) \wedge$
    $\wedge\ \forall C \in E_j.\neg N_j(C) \wedge \forall C \in E_j.(D_j(C) = dist(j,k)+1 \Rightarrow \sigma_j(C) = <>) \wedge$
    $\wedge\ queue_j = <> \wedge\ val_j = w\big)$
    (satisfied after node $j$ has recorded the argument of the received *info*-message),

  - $\big(\exists C \in E_j.(loc_j = l_{j,C,3} \wedge D_j(C) = dist(j,k)\text{-}1 \wedge\ \ \rho_j(C) = < info(w) > \wedge$
    $\wedge\ inbranch_j = C) \wedge$
    $\wedge\ \neg N_j(inbranch_j) \wedge \forall C \in E_j.(D_j(C) = dist(j,k)+1 \Rightarrow \sigma_j(C) = <>) \wedge$
    $\wedge\ queue_j = <> \wedge\ val_j = w\big)$
    (satisfied after node $j$ has recorded the identification of the edge along which the *info*-message has been received),

  - $\big(\exists C \in E_j.(loc_j = l_{j,C,4} \wedge D_j(C) = dist(j,k)\text{-}1 \wedge\ \ \rho_j(C) = < info(w) > \wedge$
    $\wedge\ inbranch_j = C) \wedge$
    $\wedge\ N_j(inbranch_j) \wedge \forall C \in E_j.(D_j(C) = dist(j,k)+1 \Rightarrow \sigma_j(C) = <>) \wedge$
    $\wedge\ queue_j = <> \wedge\ val_j = w\big)$
    (satisfied after node $j$ has recorded that it has received a message along the edge

identified by $inbranch_j$), and

- $\big(\exists C \in E_j.(loc_j = l_{j,C,5} \wedge D_j(C)=dist(j,k)\text{-}1 \wedge \ \rho_j(C)=<info(w)> \wedge$
  $\wedge\ inbranch_j = C)\wedge$
  $\wedge\ N_j(inbranch_j) \wedge \forall C \in E_j.(D_j(C)=dist(j,k)+1 \rightarrow \sigma_j(C)=<info(w)>)\wedge$
  $\wedge\ queue_j = <> \wedge val_j = w\big)$
  (satisfied after node $j$ has broadcasted the *info*-message along all adjacent edges except the one identified by $inbranch_j$).

For nodes $j$ different from $k$, the assertion $T_j^{\mathcal{B}}$ is defined as:
$T_j^{\mathcal{B}} \equiv \exists C \in E_j.loc_j = l_{j,C,5}$, which is satisfied after node $j$ has broadcasted the *info*-message along all adjacent edges except the one identified by $inbranch_j$.

Verifying the conditions (a) through (f) of appendix II for protocol $\mathcal{B}$ is straightforward, i.e., one can easily establish that $\mathcal{B}$ **sat** $< \{p_j \mid j \in V'\}, \{I_j^{\mathcal{B}} \mid j \in V'\}, \{I_j^{\mathcal{B}} \wedge T_j^{\mathcal{B}} \mid j \in V'\} >$ holds. This can, e.g., be accomplished by techniques described in [MP83]. As an example of how to prove these conditions, we shall show that condition (e) is satisfied. I.e., it must be shown that for all states in any computation sequence of $\mathcal{B}$,
(*) $I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{B}}$ ($j$ in $V$) implies that at least one action in algorithm $\mathcal{B}$ is enabled.
Below it is assumed that conditions (a) and (b) (see appendix II) have already been proven.
Choose some node $j$ in $V$.
By induction on $dist(j,k)$ we shall now show that
(**) if $I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{B}}$ holds, then there exists some node $j'$ satisfying $dist(j',k) \leq dist(j,k)$ for which at least one of its own actions is enabled.
This, obviously, implies property (*) above.
*Basis of induction:* $dist(j,k)=0$ holds. Thus, $j = k$ holds, too. Under the assumption that $I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{B}}$ holds, it follows that at least one of node $k$'s own actions is enabled. Obviously, (**) above is satified in this case.
*Induction hypothesis*: for all nodes $j$, if $I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{B}}$ and $dist(j,k)=n \geq 0$ hold, then there exists some node $j'$ satisfying $dist(j',k) \leq n$ for which at least one of its own actions is enabled.
*Induction step*: assume that $dist(j,k)=n+1$ holds. This implies that $j \neq k$ holds, too. Note that $I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{B}}$ implies that $\neg\exists C \in E_j.loc_j = loc_{j,C,5}$ is satisfied. Also, for all $C \in E_j$, $\rho_j(C)=<>$ holds, i.e., node $j$ has not received any message. If node $j$ can perform one of its actions, then

we are done, since (**) clearly holds. If node $j$ cannot perform any of its own actions, then it follows that for node $j$'s adjacent edge $C$ satisfying $D_j(C) = dist(j,k)$-1, say adjacent to node $\ell$, $\sigma_\ell(C)$=<> holds. From the invariant $I_\ell^{\mathcal{B}}$, we then obtain that $\neg T_\ell^{\mathcal{B}}$ is satisfied. (**) above now follows from the induction hypothesis and the fact that $dist(\ell, k) < dist(j, k)$ holds.

For algorithm $\mathcal{C}$ the preconditions are specified by the assertions $q_j$ ($j \in V$) defined above. The postconditions are characterized by assertions $r_j$ ($j \in V$) described by $r_j \equiv I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}}$. The assertions $I_j^{\mathcal{C}}$ and $T_j^{\mathcal{C}}$ are defined below.

For node $k$, the assertion $I_k^{\mathcal{C}}$ is the conjunction of

- $\forall n.\big(1 \le n \le \mid queue_k \mid \Rightarrow$

  $\Rightarrow \exists C \in E_k.(queue_k[n] =< ack(w), C > \wedge\ \ \rho_k(C)=< ack(w) > \wedge \wedge \neg N_k(C))\big)$

  (any element in node $k$'s queue consists of a message component $ack(w)$ and an edge component. The latter component records the identification $C$ of the edge along which the $ack$-message has been received. Moreover, $\neg N_k(C)$ holds.),

  $\forall n, m.(1 \le n < m \le \mid queue_k \mid \Rightarrow queue_k[n] \ne queue_k[m])$

  (each element in the queue is different from any other element in that queue),
- $val_k = w \wedge Tree(V, E) \wedge \mid V \mid \ge 2$,
- $\forall C \in E_k.(\ \rho_k(C) \le < ack(w) >)$

  (node $k$ can receive at most one $ack$-message along any of its adjacent edges),
- $\forall C \in E_k.(\ \sigma_k(C) = < info(w) >)$

  (node $k$ has sent an $info$-message along any of its adjacent edges),
- $\forall C \in E_k.(N_k(C) \Rightarrow \rho_k(C) = < ack(w) >)$

  (if node $k$ has recorded that it has received a message along a certain edge, then this message has been received along that edge), and
- the disjunction of
  - $\big(loc_k = at\text{-}queue_k \wedge (\neg done_k \Leftrightarrow \exists C \in E_k.\neg N_k(C))\big)$

    (satisfied initially. It also holds whenever $loc_k = at\text{-}queue_k$ is satisfied),
  - $\big(\exists C \in E_k.\big(loc_k = l_{k,C,4} \wedge \neg N_k(C) \wedge\ \ \rho_k(C) = < ack(w) > \wedge$

    $\wedge\ \forall n.(1 \le n \le \mid queue_k \mid \Rightarrow queue_k[n] \ne < ack(w), C >)\big)$

$\wedge\ \neg done_k\big)$

(satisfied after node $k$ has removed an *ack*-message from its queue),

- $\big(\exists C \in E_k.(loc_k = l_{k,C,5} \wedge N_k(C) \wedge\ \rho_k(C)=<ack(w)>) \wedge \neg done_k\big)$

  (satisfied after node $k$ has recorded the identification of the edge along which the *ack*-message has been received), and

- $(\exists C \in E_k.(loc_k = l_{k,C,6}) \wedge queue_k =<> \wedge \neg done_k \wedge \forall C \in E_k.N_k(C))$

  (satisfied after node $k$ has passed the test $\forall C \in E_k.N_k(C)$. Observe that, if this test is not passed, then the disjunct above for which $loc_k$ =*at-queue*$_k$ holds is established. The same disjunct is also established after node $k$ has performed the assignment $done_k$:=**true**.)

Notice that the assertion $I_k^{\mathcal{C}}$ is preserved whenever node $k$ receives a message.

The assertion $T_k^{\mathcal{C}}$ is defined by $T_k^{\mathcal{C}} \equiv done_k$. It holds after node $k$ has received the information that all other nodes in the network have indeed received the *info*-message.

For nodes $j$ different from node $k$, $I_j^{\mathcal{C}}$ is defined as the conjunction of

- Tree$(V, E) \wedge |V| \geq 2 \wedge val_j = w$,

  $\exists C \in E_j.(C = inbranch_j \wedge D_j(C) = dist(j,k)\text{-}1)$

  (the variable $inbranch_j$ has a defined value. The edge identified by $inbranch_j$ is the one on the shortest path from node $j$ to node $k$),
- $N_j(inbranch_j)$

  (node $j$ has recorded that it has received a message along the edge identified by $inbranch_j$),
- $\forall n.\big(1 \leq n \leq |\,queue_j\,| \Rightarrow \exists C \in E_j.\big(queue_j[n] =<ack(w), C> \wedge \neg N_j(C) \wedge$

  $$\rho_j(C)=<ack(w)>\big)\big)$$

  (cf. $I_k^{\mathcal{B}}$ above),
- $\forall n, m.(1 \leq n < m \leq |\,queue_j\,| \Rightarrow queue_j[n] \neq queue_j[m])$

  (cf. $I_k^{\mathcal{B}}$ above),
- $\forall C \in E_j.(C \neq inbranch_j \Rightarrow \sigma_j(C)=<info(w)>)$

  (node $j$ has transmitted an *info*-message along all its adjacent edges different from the edge identified by $inbranch_j$),

- $\forall C \in E_j.(C \neq inbranch_j \Rightarrow \rho_j(C) \leq < ack(w) >)$

  (node $j$ can receive at most one *ack*-message along its adjacent edges different from the edge identified by $inbranch_j$),

- $\rho_j(inbranch_j) =< info(w) >$

  (node $j$ has received an *info*-message along the edge identified by $inbranch_j$),

- $\sigma_j(inbranch_j) \leq < ack(w) >$

  i.e., node $j$ sends at most one *ack*-message along the edge identified by $inbranch_j$,

- $\forall C \in E_j.\big((N_j(C) \wedge C \neq inbranch_j) \Rightarrow \rho_j(C) =< ack(w) >$

  (for all edges $C$ different from the edge identified by $inbranch_j$ the following holds: if node $j$ has recorded that it has indeed received a message along $C$, then $j$ has received an *ack*-message along $C$), and

- the disjunction of

  - $\exists C \in E_j.(loc_j = l_{j,C,5} \wedge C = inbranch_j) \wedge \sigma_j(inbranch_j) =<> \wedge$
    $\wedge\ \forall C \in E_j.(C \neq inbranch_j \Rightarrow \neg N_j(C))$

    (satisfied initially),

  - $\big(\exists C \in E_j.(loc_j = l_{j,C,6} \wedge C = inbranch_j) \wedge \sigma_j(inbranch_j) =<>\big) \wedge$
    $\wedge\ \forall C \in E_j.(C = inbranch_j) \wedge queue_j =<>\big)$

    (satisfied after node $j$ has passed the test $\forall C \in E_j.N_j(C)$),

  - $(loc_j = at\text{-}queue_j)$

    (satisfied after node $j$ has transmitted the *ack*-message along the edge identified by $inbranch_j$),

  - $\big(\exists C \in E_j.\big(loc_j = l_{j,8,C} \wedge D_j(C) = dist(j,k)+1 \wedge \neg N_j(C) \wedge$
    $\wedge\ \forall n.(1 \leq n \leq |\ queue_j\ | \Rightarrow queue_j[n] \neq < ack(w), C >)\big)\big)$

    (satisfied after node $j$ has removed an *ack*-message from its queue),

  - $\big(\exists C \in E_j.(loc_j = l_{j,9,C} \wedge D_j(C) = dist(j,k)+1 \wedge N_j(C)\big)$

    (satisfied after node $j$ has recorded that it has received a message along the edge identified by the edge component of the most recently removed message from the queue), and

  - $\big(\exists C \in E_j.(loc_j = l_{j,10,C} \wedge D_j(C) = dist(j,k)+1 \wedge \forall C \in E_j.N_j(C))\big)$

    (satisfied after node $j$ has passed the test $\forall C \in E_j.N_j(C)$. Observe that if this test is not passed or if an ack-message is transmitted by node $j$ along the edge identified by $inbranch_j$, then the the assertion $I_j^C$ is preserved. It is also preserved if node $j$ receives an ack-message.).

For $j \neq k$ we define $T_j^C$ as

$T_j^C = \sigma_j(inbranch_j) - < ack(w) > \wedge \forall C \in E_j.N_j(C)$. It holds after node $j$ has sent a message along the edge identified by $inbranch_j$.

It can be shown that $C$ **sat** $\{I_j^B \wedge T_j^B \mid j \in V'\}, \{I_j^C \mid j \in V'\}, \{I_j^C \wedge T_j^C \mid j \in V'\} >$ holds (cf. appendix II).

Establishing the verification conditions (3) through (6) formulated in appendix II is straightforward. Obviously, verification condition (3) is true. As an example of how one could establish the other conditions, we shall show how the first disjunct of condition (4) can be shown to hold for node $k$, i.e., we shall show that *disabled*$(I_k^B \wedge \neg T_k^B, IS_k^C)$ holds.
In order to do so, notice that if $I_k^B \wedge \neg T_k^B$ holds, then an action in the set $IS_k^C$ can be enabled only if $loc_k$—*at-queue*$_k$ is satisfied. The latter implies that only actions by which an *ack*-message is removed from node $k$'s message queue can be enabled. $I_k^B \wedge \neg T_k^B$ implies, however, that $k$'s message queue cannot contain any *ack*-messages.

# CHAPTER 4

# A detailed analysis of Gallager, Humblet, and Spira's distributed minimum-weight spanning tree algorithm

## –An example of sequentially phased reasoning–

F.A. Stomp
University of Nijmegen, Department of Computer Science,
Toernooiveld, 6525 ED Nijmegen, The Netherlands.
Email address: frank@cs.kun.nl.

W.P. de Roever
Eindhoven University of Technology,
Department of Mathematics and Computing Science,
POB 513, 5600 MB Eindhoven, The Netherlands.
Email address: wsinwpr@eutrc3.urc.tue.nl.

**Abstract:** Correctness of the distributed minimum-weight spanning tree algorithm of Gallager, Humblet, and Spira [GHS83] is proved. Two kinds of (slight) optimizations w.r.t. the number of transmitted messages during execution of the algorithm are proposed. A source of failure of the algorithm is detected and corrected. The correctness proof exemplifies our principle for sequentially phased reasoning about concurrent programs [SR89a, SR89b]. Our proof illustrates that correctness proofs of complex algorithms can be structured according to their designers' intuition.

# 1 Introduction

Ever since Floyd [F67] proposed his method for verifying (sequential) programs, represented by means of flowcharts, various proof methods have been presented in the literature [AFR80, H69, L83, MC81, OG76, ZRE85, Z89], for reasoning about sequential and distributed programs.

Proof methods can, in general, be classified as *compositional* ones, such as those in [H69, L83, MC81, ZRE85] and in [Z89], in which the specification of a program is verified on the basis of specifications of its constituent components without referring to the internal construction of those components [Z89], and as *non-compositional* ones, such as those in [AFR80, F67, OG76].

Examples of the applicability of the latter mentioned verification methods illustrate, almost without exception, that *the reasoning about a program takes place after that program has been constructed.*

The technique of transformational programming [BK83, CM88, D76, P89] has also received a lot of attention. This technique advocates deriving a program, starting from some formal specification, by successively applying correctness preserving transformation principles. The program, thus obtained, satisfies (by definition) the initial specification. As a consequence, the technique of transformational programming can be viewed as a verification technique, *where the program to be proved correct is derived, or constructed, during its verification phase.* It enables one *to develop a program and its proof hand-in-hand, with the proof ideas leading the way* [G81].

Recently, we have proposed in [SR89a, SR89b] a transformation principle for *sequentially phased* reasoning about *concurrently* performed (sub)tasks in network algorithms. That is, if a certain task to be performed by processes in some network can be split up, from a logical point of view, into several subtasks as if they are performed sequentially, then our principle describes how one can combine the programs solving the subtasks in order to obtain one program which solves the whole task. (Viewed as a proof principle in some proof system, any such proof system is a non-compositional one.) From an analyzer's or from a designer's point of view this kind of decomposition of a task into subtasks is quite attractive, since it allows him to concentrate on a single subject at a time.

A large number of complex network algorithms, such as those for minimum-path, connectivity, network flow, and minimum-weight spanning trees described in [Hu83, MS79, Se82, Se83, ZS80], are structured according to our principle.

*As shown in the present paper, the complicated distributed minimum-weight spanning tree algorithm* *of Gallager Humblet, and Spira* [GHS83] *is also structured according to this principle.*

Probably the simplest network algorithm in which one may decompose the design of a program, or the reasoning about it, into subprograms as if they are performed sequentially is Segall's PIF-protocol [Se83], also see [DS80] and [Fr80], which is a broadcasting protocol. In this algorithm, the whole task performed by the processes in a certain network can be described as follows: Some value $w$, initially recorded by some process $k$ is supplied to all other processes in the network, and $k$ is informed that all nodes have recorded this value indeed. This task can be decomposed into two subtasks as if they are performed sequentially: the *first subtask* broadcasting the value $w$, and the *second one* reporting back that the processes in the network have received and recorded $w$.
The same kind of decomposition can also be discerned in the distributed minimum-weight spanning tree algorithm of Gallager, Humblet, and Spira [GHS83], which will from now on be abbreviated to Gallager's algorithm. Here one may decompose the whole task of constructing the minimum-weight spanning tree of a network into five (sub)tasks. Apart from the fact that these five tasks are performed sequentially from a logical point of view, that algorithm displays other additional features (see section 6): *expanding* groups of nodes perform the five tasks *repeatedly*, with different groups of nodes performing these tasks *concurrently* w.r.t. another, and a certain task performed by one group of nodes can be *disturbed temporarily* due to interference with the task of another group.
We define two other principles for coping with these additional features: One principle describes how to combine programs which are executed completely independent of each other, i.e., when programs are executed concurrently w.r.t. another and no communication occurs between two distinct programs. The second principle describes how to deal with the above-mentioned kind of interference.

As argued in the sections 4, 5, and 6, the (distributed) program describing Gallager's algorithm, which will from now on be abbreviated to Gallager's program, can be derived from a sequential program which constructs the minimum-weight spanning tree of a graph. That is, one can start with a sequential program that constructs the minimum-weight spanning tree of a graph, then refine parts of this program until distributed programs are obtained (each such part corresponds to some description how a certain task can be solved), and finally combine by means of our principles the distributed programs found above into one program. The final (distributed) program, thus obtained, is Gallager's. This particular strategy has allowed us to find two (slight) optimizations of the program in [GHS83] w.r.t. the number of message transmitted when executing Gallager's program. We have, in addition, as a consequence of our kind of reasoning, detected that the program in [GHS83] does not necessarily construct the minimum-weight spanning trees for arbitrary graphs. (The reason for this is explained in section 6.)

The first attempt to prove correctness of Gallager's algorithm appears in [SR87]. The proof there is based on the above-mentioned kind of decompositions of tasks into subtasks. There the principle for sequential phased reasoning has been identified as an independent principle, but this principle has not been formulated nor justified. Consequently, the proof in [SR87] should be considered incomplete. Welch, Lamport, and Lynch [WLL88a] have given a correctness proof of Gallager's algorithm using a partial hierarchy of algorithms. Unfortunately, their complete proof is a very lengthy one, cf. [WLL88b]. Chou and Gafni [CG88] have analyzed a minimum-weight spanning tree algorithm of which they claim that it is a simplified version of Gallager's. They have, however, not verified Gallager's algorithm. (In fact, they have verified a far much simpler algorithm than Gallager's, cf. section 4.)

The remainder of this paper is organized as follows: in section 2 we introduce some notation used in this paper. We describe our principle for sequentially phased reasoning about concurrently performed (sub)tasks in section 3. The basic features of Gallager's algorithm and of its correctness proof are the subjects of section 4. In section 5 the formal specification is presented which Gallager's program should satisfy. In section 6 it is shown that this is the case indeed. Finally, section 7 contains some conclusions.

## 2 Preliminaries

In this section some notations and conventions, used throughout this paper, are introduced.

The reader is assumed to be familiar with elementary notions from graph-theory, such as graphs, trees, and cycles, and with their definitions and properties (cf. [E79]). Graphs are denoted by tuples $(V, E)$ consisting of a set of nodes $V$ and a set of edges $E$. For graphs $(V_1, E_1)$ and $(V_2, E_2)$, $(V_1, E_1)$ is called a subgraph of $(V_2, E_2)$, denoted by $(V_1, E_1) \subseteq (V_2, E_2)$, iff $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$ are both satisfied. If $(V_1, E_1) \subseteq (V_2, E_2)$ holds and if $(V_1, E_1)$ constitutes a tree, then $(V_1, E_1)$ is called a subtree of $(V_2, E_2)$. The graphs $(V_1, E_1)$ and $(V_2, E_2)$ are distinct, denoted by $(V_1, E_1) \neq (V_2, E_2)$, iff $V_1 \neq V_2$ or $E_1 \neq E_2$ is satisfied. In the sequel $i$, $j$, and $k$, possibly primed or indexed, will denote nodes; edges will be denoted by $e$ and $e'$. For a graph $(V, E)$ and a node $i$ in $V$, the set of all edges adjacent to node $i$ will be denoted by $E_i$. Hereafter, $E_{i,j}$ will abbreviate the set $E_i \cap E_j$, i.e., $E_{i,j}$ denotes the set of all edges connecting the nodes $i$ and $j$ $(i, j \in V)$.

The distributed algorithms considered in this paper are performed by processes in a fixed, finite, and undirected network which will be represented by a graph $(V, E)$. Processes are identified with nodes in $V$; Communication channels are identified with edges in $E$. Adjacent nodes communicate by means of messages. Since edges are undirected, each node can both send and receive messages along any of its adjacent edges. Communication is *asynchronous*, i.e., messages transmitted by some node along one of its adjacent edges always arrive within a finite, but unpredictable, time frame at the other end of that edge. Communication is assumed to be *perfect*, i.e., messages transmitted by some node along one of its adjacent edges arrive in sequence, error-free, without loss, and without duplication at the other end of that edge.

# 3 Our proof principle for sequentially phased reasoning

We now present our proof principle which states that one can reason *sequentially* about *concurrently* performed (sub)tasks. For a fully worked out illustration, applied to Segall's PIF-protocol [Se83], the reader is referred to [SR89b].

## 3.1 Notation

We consider distributed algorithms which are performed by nodes in a network $(V, E)$. A distributed algorithm $\mathcal{D}$ is represented by a triple $< V', \{p_i \mid i \in V'\}, Act^{\mathcal{D}} >$. $V' \subseteq V$ denotes the set of nodes containing all those nodes that actually execute the algorithm; This implies that if some node in the set $V'$ sends a message along one of its adjacent edges $e$ when it executes algorithm $\mathcal{D}$, then the node at the other end of $e$ is in $V'$, too. $p_i$ (node $i$'s precondition) is a state assertion characterizing the initial values of node $i$'s variables and the initial contents of node $i$'s adjacent edges; $Act^{\mathcal{D}}$ is a set of (atomic) actions containing all those actions which can occur in any computation sequence of the algorithm (cf. definition 3.1 below). Each action $a$ in the set $Act^{\mathcal{D}}$ has some enabling condition $en(a)$ associated with it. Such a condition consists of a boolean expression or of a boolean expression combined with a receive-statement (cf. [H78]). (In the technical formulation of our principle, see section 3.3, the boolean part of the enabling condition of action $a$ will be denoted by $bp(a)$.) Moreover, the set $Act^{\mathcal{D}}$ can be partitioned into sets $Act_i^{\mathcal{D}}$ such that each $Act_i^{\mathcal{D}}$ consists of all actions which can be executed by node $i$ ($i \in V'$).

**Definition 3.1** Let $\mathcal{D} = < V', \{p_i \mid i \in V'\}, Act^{\mathcal{D}} >$ be an algorithm. A computation sequence of $\mathcal{D}$ is a maximal sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a2} \cdots$ such that for all $n \geq 0$ the following is satisfied: $s_n$ is a state, each $p_i$ ($i \in V'$) holds in state $s_0$, $a_n$ occurs in the set $Act^{\mathcal{D}}$, action $a_n$ is enabled in state $s_n$, i.e., $a_n$'s enabling condition holds in $s_n$, and $s_{n+1}$ is the state resulting when action $a_n$ is executed in state $s_n$. (As usual, a computation sequence is considered to be maximal if it is infinite, or if it is finite and no action in the set $Act^{\mathcal{D}}$ is enabled in the last state of the sequence.) ■

The reason for allowing the first component $V'$ in the triple above to be a proper subset of $V$, i.e., the set of all nodes in the network, is that in Gallager's algorithm the tasks which we analyze are not performed by a fixed group of nodes. More precisely, these tasks are performed by dynamically changing groups of nodes. As a consequence, we explicitly indicate in an algorithm which nodes may actually be involved in the execution of an algorithm.

We conclude this subsection with the following:

**Definition 3.2** Let $j$ be some node in $V'$. ($V'$ denotes the first component in algorithm $\mathcal{D}$, see above). Let $e$ denote some edge adjacent to node $j$ and to another node $i$ in the set $V'$.

(a) $Int_j^{\mathcal{D}} \subseteq Act_j^{\mathcal{D}}$ denotes the set of node $j$'s internal actions.

(b) $Rec_j^{\mathcal{D}}(e) \subseteq Act_j^{\mathcal{D}}$ denotes the set of node $j$'s actions which involve the receipt of a message along edge $e \in E_j$.

(c) $Sen_j^{\mathcal{D}}(e) \subseteq Act_j^{\mathcal{D}}$ denotes the set of node $j$'s actions which involve the transmission of a message along edge $e \in E_j$.

(d) Hereafter $IS_j^{\mathcal{D}}$ will denote the set of node $j$'s internal actions and those actions which involve the transmission of a message, i.e., $IS_j^{\mathcal{D}} = Int_j^{\mathcal{D}} \cup \bigcup_{e \in E_j} Sen_j^{\mathcal{D}}(e)$. ■

It is assumed that for each algorithm $\mathcal{D}$ as above the set $Act_j^{\mathcal{D}}$ can be partitioned into the (possibly empty) sets $Int_j^{\mathcal{D}}$, $Sen_j^{\mathcal{D}}(e)$, and $Rec_j^{\mathcal{D}}(e)$ ($j \in V'$, $e \in E_{i,j}$ for some node $i \in V'$).

### 3.2 Correctness formulae

Let $\mathcal{D} = < V', \{p_i \mid i \in V'\}, Act^{\mathcal{D}} >$ be an algorithm for which the the following should hold: Every finite computation sequence of $\mathcal{D}$ ends in a state satisfying some (given) state assertions $q_i$ ($i \in V'$). I.e., algorithm $\mathcal{D}$ is supposed to solve a (sub)task described by the pair of state assertions $\{p_i \mid i \in V'\}$ (the preconditions) and $\{q_i \mid i \in V'\}$ (the postconditions).

We now introduce correctness formulae of the form
$\mathcal{D}$ **sat** $< \{I_j \mid j \in V'\}, \{T_j \mid j \in V'\}, \{q_j \mid j \in V'\} >$. Here $I_j$, $T_j$, and $q_j$ are state assertions. A correctness formula as above is valid if for every computation sequence of $\mathcal{D}$ the following hold:

- For all $j \in V'$, $I_j$ holds in each state of the sequence.
- For all $j \in V'$, $T_j$ holds iff node $j$ will not execute any action in $Act^{\mathcal{D}}$ anymore. $T_j$ is called node $j$'s *termination condition*.
- For all $j \in V'$, $q_j$ holds when and if node $j$ has completed its participation in $\mathcal{D}$.

A correctness formula as above can be characterized in linear time temporal logic [MP83]. This is the subject of definition 3.2 below. We have used there, as in [SS84], *auxiliary proof variables* $\sigma_j(e)$ and $\rho_j(e)$ (for nodes $j \in V'$ and for edges $e \in E_j$). They are used for reasoning about communication. $\sigma_j(e)$ records the sequence of all messages transmitted by node $j$ along edge $e$; $\rho_j(e)$ records the sequence of all messages received by node $j$ along edge $e$. For nodes $i$ and $j$ and for edges $e \in E_{i,j}$, the property $\rho_j(e) \leq \sigma_i(e)$ is preserved by any action, see [SS84]. That is, if edge $e$ connects the nodes $i$ and $j$, then the sequence of all messages received by node $j$ along edge $e$ is a prefix of all messages transmitted by node $i$ along edge $e$. These auxiliary proof variables are changed when a node transmits or receives a message; they are not changed during the execution of an internal action. (An internal action does not involve any communication between nodes.)
For a certain node $j \in V'$ and for a certain edge $e$ adjacent to $j$, action $a \in Rec_j^{\mathcal{D}}(e)$ is enabled (recall that $Rec_j^{\mathcal{D}}(e)$ has been introduced in definition 3.2) iff the following holds: the boolean part $bp(a)$ of action $a$'s enabling condition is true and the sequence of all messages received by node $j$ along edge $e$ is a proper prefix of the sequence of all messages transmitted by the node at the other end of edge $e$.

Formally, for such an action $en(a)$ holds iff $bp(a) \wedge \rho_j(e) < \sigma_k(e)$ is satisfied where $k$ is some node in $V'$ such that $e \in E_{j,k}$.

Of course, for any action $a \in IS_j^{\mathcal{C}}$ the enabling condition $en(a)$ of $a$ is the same as the boolean part of this enabling condition, i.e., $en(a) = bp(a)$ is satisfied.

**Definition 3.3** The correctness formula $\mathcal{D}$ **sat** $< \{I_j \mid j \in V'\}, \{T_j \mid j \in V'\}, \{q_j \mid j \in V'\}$, cf. above, is an abbreviation of the conjunction of the conditions (a) through (f) below. (Some of these conditions are redundant. They have been included in order to formalize the intuition behind such a correctness formula in a natural way). The conditions are interpreted over all computation sequences of $\mathcal{D}$. (Below $\Box$ denotes the always-operator from temporal logic.)

(a) $\forall j \in V'.\Box(pre_j^{\mathcal{C}} \Rightarrow I_j^{\mathcal{C}}) \wedge \forall j,k \in V'.\forall e \in E_{j,k}.\Box(pre_j^{\mathcal{D}} \Rightarrow \rho_j(e) \leq \sigma_k(e))$.

That is, initially the assertion $I_j^{\mathcal{D}}$ holds for all nodes $j$ in $V'$. Furthermore, the sequence of all messages received by a certain node along any of its adjacent edges is a prefix of the sequence of all messages transmitted by the node at the other end of that edge is satisfied initially. (From the discussion above it follows that the property $\forall j,k \in V'.\forall e \in E_{i,j}.\rho_j(e) \leq \sigma_k(e)$ continuously holds during execution of algorithm $\mathcal{D}$.)

(b) $\forall j \in V'.\Box\big((I_j^{\mathcal{D}} \wedge \neg T_j^{\mathcal{D}}) U (I_j^{\mathcal{D}} \wedge T_j^{\mathcal{D}})\big)$ . Here $U$ denotes the weak-until operator, cf. [MP83].

We thus have that $I_j^{\mathcal{D}}$ is an invariant and for all computation sequences of $\mathcal{D}$ "node $j$ participates in the algorithm until it has completed its participation".

(c) $\forall j \in V'.\forall a \in Act_j^{\mathcal{D}}.\Box((I_j^{\mathcal{D}} \wedge T_j^{\mathcal{D}}) \Rightarrow \neg en(a))$.

(For actions $a$, $en(a)$ has been defined above.) I.e., if a certain node has completed its participation in algorithm $\mathcal{D}$, then it cannot perform any action associated with $\mathcal{D}$ anymore.

(d) $\forall j \in V'.\Box((I_j^{\mathcal{D}} \wedge T_j^{\mathcal{D}}) \Rightarrow \Box(I_j^{\mathcal{D}} \wedge T_j^{\mathcal{D}}))$ .

That is, once a node has completed its participation in $\mathcal{D}$, then it will never participate in the algorithm anymore.

(e) $\forall j \in V'.\Box\big((I_j^{\mathcal{D}} \wedge \neg T_j^{\mathcal{D}}) \Rightarrow (\exists a \in Act^{\mathcal{D}}.en(a))\big)$.

If a certain node has not completed its participation in algorithm $\mathcal{D}$, then $\mathcal{D}$ cannot be completed, i.e., at least one action in $Act^{\mathcal{D}}$ is enabled.

(f) $\forall j \in V'.\Box((I_j^{\mathcal{D}} \wedge T_j^{\mathcal{D}}) \Rightarrow post_j^{\mathcal{D}})$.

I.e., if node $j$ has completed its participation in $\mathcal{D}$, then $j$'s postcondition holds. ■

### 3.3 Description of the proof principle

Let $\mathcal{A}$=< $V'$, $\{pre_i^{\mathcal{A}} \mid i \in V'\}$, $Act^{\mathcal{A}}$ > and $\mathcal{B}$=< $V'$, $\{pre_i^{\mathcal{B}} \mid i \in V'\}$, $Act^{\mathcal{B}}$ > be two algorithms. Let $\mathcal{A}$ solve the subtask described by the pair of assertions $\{pre_j^{\mathcal{A}} \mid i \in V'\}$, $\{pre_j^{\mathcal{B}} \mid i \in V'\}$. Let $\mathcal{B}$ solve the subtask described by the pair of assertions $\{pre_j^{\mathcal{B}} \mid i \in V'\}$, $\{post_j^{\mathcal{B}} \mid i \in V'\}$. Assume that we have shown that for certain state assertions $I_j^{\mathcal{A}}$, $I_j^{\mathcal{B}}$, $T_j^{\mathcal{A}}$, and $T_j^{\mathcal{B}}$ ($j \in V'$)

(1) $\mathcal{A}$ **sat** $< \{I_j^{\mathcal{A}} \mid j \in V'\}, < \{T_j^{\mathcal{A}} \mid j \in V'\}, \{pre_j^{\mathcal{B}} \mid j \in V'\} >$ and

(2) $\mathcal{B}$ **sat** $< \{I_j^{\mathcal{B}} \mid j \in V'\}, < \{T_j^{\mathcal{B}} \mid j \in V'\}, \{post_j^{\mathcal{B}} \mid j \in V'\} >$

both hold. If the *verification conditions* (3) through (6) below hold, too, then the algorithm consisting of all actions occurring in $\mathcal{A}$ and $\mathcal{B}$ solves the task described by $\{pre_j^{\mathcal{A}} \mid i \in V'\}$ and $\{post_j^{\mathcal{B}} \mid i \in V'\}$. Moreover, for all $j$ in $V'$, $I_j^{\mathcal{A}} \vee I_j^{\mathcal{B}}$ is an invariant of this algorithm.
More precisely, if all the conditions (1) through (6) are satisfied, then for the algorithm $\mathcal{C}$=< $V'$, $\{pre_i^{\mathcal{A}} \mid i \in V'\}$, $Act^{\mathcal{A}} \cup Act^{\mathcal{B}}$ >, $\mathcal{C}$ **sat** $< \{I_j^{\mathcal{A}} \vee I_j^{\mathcal{B}} \mid j \in V'\}, < \{I_j^{\mathcal{B}} \wedge T_j^{\mathcal{B}} \mid j \in V'\}, \{post_j^{\mathcal{B}} \mid j \in V'\} >$ holds.

As a preparation for the technical formulation of the verification conditions (3) through (6) below, we first introduce an auxiliary assertion.

**Definition 3.4** Let $P$ denote some state assertion. Let $AC$ be a certain set of actions.
Define the assertion *disabled*($P$, $AC$) by *disabled*($P$, $AC$) $\equiv \Box(P \Rightarrow \forall a \in AC.\neg en(a))$.
Thus, *disabled*($P$, $AC$) expresses that if assertion $P$ holds, then all actions in $AC$ are disabled. ■

The following conditions are required for a sound application of our principle:

(3) Each of the programming variables occurring in $pre_j^{\mathcal{A}}$, $pre_j^{\mathcal{B}}$, $post_j^{\mathcal{B}}$, $I_j^{\mathcal{A}}$, $I_j^{\mathcal{B}}$, $T_j^{\mathcal{A}}$, and $T_j^{\mathcal{B}}$ is node $j$'s own variable. If the proof variables $\rho_\ell(e)$ or $\sigma_\ell(e)$ occur in any of these assertions, then $\ell$=$j$ and $e \in E_j$ are satisfied. (Variables occurring in any of the above assertions can be changed only as a result of the execution of one of node $j$'s actions.)

(4) $\forall j \in V'.disabled(I_j^{\mathcal{A}} \wedge \neg T_j^{\mathcal{A}}, IS_j^{\mathcal{B}}) \wedge \forall j,k \in V'.\forall e \in E_{j,k}.disabled(I_j^{\mathcal{A}} \wedge \neg T_j^{\mathcal{A}}, Sen_k^{\mathcal{B}}(e))$ holds for all computation sequences of $\mathcal{A}$.

This condition states that if a certain node has not completed its participation in algorithm $\mathcal{A}$, then it can perform neither an internal action nor a send-action occurring in algorithm $\mathcal{B}$ (the first conjunct), and it cannot receive a message associated with algorithm $\mathcal{B}$ (the second conjunct). The latter is satisfied because if the node participates in algorithm $\mathcal{A}$, then it is required that none of its neighbors can send such messages. Consequently, this condition ensures that if a certain node has not completed its participation in algorithm $\mathcal{A}$, then it cannot perform any of its actions associated with $\mathcal{B}$.

Of course, we also require that no node can perform any action associated with algorithm $\mathcal{A}$, if it is participating in algorithm $\mathcal{B}$:

(5) $\forall j \in V'.disabled(I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{B}}, IS_j^{\mathcal{A}}) \wedge \forall j,k \in V'.\forall e \in E_{j,k}.disabled(I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{B}}, Sen_k^{\mathcal{A}}(e))$ holds for all computation sequences of $\mathcal{B}$.

(6) $\forall j \in V'.disabled(I_j^{\mathcal{B}} \wedge T_j^{\mathcal{B}}, IS_j^{\mathcal{A}}) \wedge \forall j,k \in V'.\forall e \in E_{j,k}.disabled(I_j^{\mathcal{B}} \wedge T_j^{\mathcal{B}}, Sen_k^{\mathcal{A}}(e))$ holds for all computation sequences of $\mathcal{B}$.

Therefore, each node which participates in both subtasks participates in the first subtask, i.e., the one solved by algorithm $\mathcal{A}$, before it participates in the second subtask, i.e., the one solved by algorithm $\mathcal{B}$.

If, in addition, one wants to prove that the algorithm solving the whole task always terminates, then it suffices to prove that both the algorithms $\mathcal{A}$ and $\mathcal{B}$ always terminate. An algorithm $\mathcal{D}\in\{\mathcal{A}, \mathcal{B}\}$ as above terminates iff for all $j \in V'$, $\Diamond(I_j^{\mathcal{D}} \wedge T_j^{\mathcal{D}})$ holds for all computation sequences of $\mathcal{D}$. Here, $\Diamond$ denotes the eventual-operator from temporal logic.

How to reason, according to this strategy, about an algorithm which solves a task that can be split up logically into more than two subtasks, as if they are performed sequentially, should be obvious. (This is a straightforward extension of the case treated above, cf. [SR89a, SR89b]; It can also be achieved by repeatedly applying the above principle.)

# 4 Basic features of Gallager's algorithm and of its correctness proof

Gallager's algorithm is a distributed algorithm for constructing the minimum-weight spanning tree of a finite, undirected, and connected graph $(V, E)$ in which each edge in $E$ has some strictly positive *weight* associated with it, such that distinct edges have distinct weights. In section 4.1 we present two theorems, well-known from graph-theory, upon which the correctness of Gallager's algorithm is based. The essentials of this algorithm are described in section 4.2. The structure of our correctness proof is presented in section 4.3. The discussion in this section shows that both structured verification and structured design of complex algorithms can be achieved by decomposing the reasoning and the design of such an algorithm according to its *logical (sub)tasks*.

## 4.1 Theorems underlying the correctness of Gallager's algorithm

Let $(V, E)$ be a finite, undirected, and connected graph $(V, E)$. Assume that $w: E \rightarrow \mathbb{R}^+$ is a function assigning weights to edges, where $\mathbb{R}^+$ denotes the set of all real numbers greater than 0. Furthermore, assume that $w$ is an injective function, i.e., that distinct edges have distinct weights. From now on, such weighted graphs will be denoted by $(V, E, w)$.

Correctness of Gallager's algorithm is based on the *existence* and the *uniqueness* of the minimum-weight spanning tree of any such graph as above.

**Theorem 4.1** Given any weighted graph $(V, E, w)$. There exists a unique minimum-weight spanning tree of $(V, E)$.

**Proof:** The existence of at least one minimum-weight spanning tree of the weighted graph should be clear. To show the uniqueness of the spanning tree, we assume, in order to obtain a contradiction, that there exist two spanning such spanning trees $T_1$ and $T_2$ satisfying $T_1 \neq T_2$. Then, obviously, there exists an edge occurring in one, but not in both these trees. Let $e$ be the minimum-weight such edge. W.l.o.g. assume that edge $e$ occurs in $T_1$ and not in $T_2$. Now, consider the graph obtained by adding edge $e$ to the tree $T_2$. This graph contains a cycle. It follows that at least one edge $e'$ on this cycle does not occur in the tree $T_1$, since $T_1$ is free of cycles. Note that $e \neq e'$ holds. Moreover, $w(e') < w(e)$ holds, too. (Otherwise, removing edge $e'$ from the tree $T_2$ and adding $e$ to $T_2$ would yield a spanning tree of the weighted graph $(V, E, w)$ with less weight than $T_2$, contradicting that $T_2$ is a minimum-weight spanning tree of $(V, E, w)$.) Removing edge $e$ from the tree $T_1$ and adding edge $e'$ to $T_1$ then yields a spanning tree of the graph $(V, E, w)$ with less weight than $T_1$. This contradict the assumption that $T_1$ is a minimum-weight spanning tree of $(V, E, w)$. We conclude that there exists exactly one minimum-weight spanning tree of $(V, E, w)$. ■

Theorem 4.1 ensures the existence of a unique minimum-weight spanning tree of a weighted graph. How one could actually construct this tree is suggested by theorem 4.2 below. As a preparation for this theorem we define two notions that will be used extensively in the remainder of this paper.

**Definition 4.1** Given a weighted graph $(V, E, w)$ as above. Denote by $T$ the (unique) minimum-weight spanning tree of that graph.

(a) A *fragment of* $T$ is some non-empty subtree of $T$.

(b) Assume that $F=(V', E')$ is some fragment of $T$. An edge $e \in E$ is an *outgoing edge of* $F$ iff one of the nodes adjacent to $e$ is in $F$ and the other one is not. In other words, edge $e$ is an outgoing edge of $F$ iff the following is satisfied: for nodes $i$ and $j$ satisfying $e \in E_{i,j}$, $(i \in V' \wedge j \notin V') \vee (i \notin V' \wedge j \in V')$ holds. (Cf. section 2 for the interpretation of the sets $E_{i,j}$.) ■

We then have the following

**Theorem 4.2** Let $F'=(V', E')$ and $F''=(V'', E'')$ be two disjoint fragments of the minimum-weight spanning tree $T$ of a weighted graph $(V, E, w)$.

(a) If $e \in E$ is the minimum-weight outgoing edge of $F'$ and $e$ is adjacent to $F''$, i.e., adjacent to some node in $F''$, then $F'''=(V' \cup V'', E' \cup E'' \cup \{e\})$ is a fragment of $T$, too.

(b) $T=F'$ iff no outgoing edge of $F'$ exists.

**Proof:**

(a) Suppose, in order to obtain a contradiction, that $F'''$ is not a fragment of $T$. Consequently, edge $e$ is not in the tree $T$. By an argument analogous to the one in theorem 4.1, this leads to a contradiction.

(b) Clearly, $T=F'$ implies that there are no outgoing edges of $F'$. In order to prove the other implication, assume that there exists no outgoing edge of $F'$. Suppose, in order to obtain a contradiction, that $T \neq F'$ is satisfied. It then follows that there exists an edge $e$ occurring in $T$ and not in $F'$. As above, the existence of such an edge leads to a contradiction. ■

## 4.2 High-level description of Gallager's algorithm

From now on we assume some fixed weighted graph ($V$, $E$, $w$). The minimum-weight spanning tree of this graph will be denoted by $T$.

A large number of algorithms, both sequential and distributed ones, have been suggested by theorem 4.2 (see, e.g., [D59, GHS83, K56, ZS80]). All these algorithms have *in common* that they start with trivial fragments of $T$, consisting of a single node (and, thus, without any edges), and gradually enlarge these fragments as described in theorem 4.2 until $T$ has been constructed. The algorithms *differ* in *how* and *when* fragments are enlarged. E.g., the algorithms reported in [D59, ZS80] start with one particular trivial fragment and gradually enlarge this fragment with one node and one edge at a time. The algorithm reported in [K56] starts with all trivial fragments. Two fragments combine if they have the same minimum-weight outgoing edge and this edge has the least weight among all outgoing edges of the fragments constructed so far.

Gallager's algorithm also starts with all trivial fragments in the graph. Fragments are combined into larger ones according to a more sophisticated strategy than those ones adopted in e.g., [D59, ZS80], and [K56]: *the combinations of fragments depend on so-called levels*. The *level of a fragment of* $T$ is (inductively) defined below.

**Definition 4.2**

(i) A fragment consisting of a single node, i.e., a trivial fragment, is defined to be at level 0.

Next assume that fragment $F$ is at level $L$. Let edge $e$ be $F$'s minimum-weight outgoing edge. Denote by $F'$ the fragment, say at level $L'$ at the other end of $e$. When $F$ and $F'$ are disjoint, then the following is satisfied:

(ii) If the fragments $F$ and $F'$ are at the same level, i.e., $L{=}L'$ holds, and if edge e is the minimum-weight outgoing edge of $F'$, then the fragment formed by combining $F$ and $F'$ is defined to be at level $L{+}1$ $(=L'{+}1)$.

(iii) If $L < L'$ is satisfied, then the fragment formed by combining $F$ and $F'$ is defined to be at level $L'$. ■

In Gallager's algorithm fragments only combine according to one of the possibilities (ii) and (iii) above. If neither of these possibilities apply, then, from an operational point of view, fragment $F$ simply waits until one of these two possibilities occurs. This delay does not lead to a deadlock, i.e., if a fragments waits for one of the two possibilities above to occur and the minimum-weight spanning tree has not yet been constructed, then one of the possibilities shall eventually occur. This is proved in theorem 6.1. A sequential description of Gallager's algorithm is shown in figure 1 below. Under the assumption that fair selections are made in this program, it indeed constructs the minimum-weight spanning tree.

**while** $|\mathcal{F}| \neq 1$
**do** select some $F' \in \mathcal{F}$;
  let $F' = < (V', E'), L' >$;
  let $e =$ minimum-weight outgoing edge of $(V', E')$; (1)
  let $F'' = < (V'', E''), L'' > \in \mathcal{F}$ such that $(V'', E'')$ is adjacent to $e$, $(V'', E'') \neq (V', E')$;
  **if** $L' = L'' \wedge e =$ minimum-weight outgoing edge of $(V'', E'')$
  **then** $\mathcal{F} := \mathcal{F} - \{F', F''\} \cup \{< (V' \cup V'', E' \cup E'' \cup \{e\}), L' + 1 >\}$
  **elif** $L' < L''$ (2)
  **then** $\mathcal{F} := \mathcal{F} - \{F', F''\} \cup \{< (V' \cup V'', E' \cup E'' \cup \{e\}), L'' >\}$
  **fi**
**od**

**Notation:** $\mathcal{F}$ is a collection of pairs containing a fragment $(V', E')$ of $T$ as its first component and containing the level of $(V', E')$ as its second component. $|\mathcal{F}|$ denotes the cardinality of $\mathcal{F}$. Initially, $\mathcal{F}$ consists of all trivial fragments having 0 as their level, i.e., $\mathcal{F} = \{<(\{i\}, \emptyset), 0> \mid i \in V\}$ holds.

Figure 1. A sequential version of Gallager's algorithm.

In the algorithms reported in [D59] and [ZS80] essentially *one* fragment is enlarged by appending its minimum-weight outgoing edge and one node adjacent to this edge, until $T$ has been constructed. As such, constructing $T$ is restricted to a rather strong requirement, not taking into account that many fragments could be combined into larger fragments independently of other ones. In Kruskal's algorithm [K56], however, *many* fragments could be combined into larger ones independently from each other. Yet, fragments are combined only if they have the same minimum-weight outgoing edge. (Although Chou and Gafni [CG88] have claimed that they have proved the correctness of Gallager's algorithm, they have, in fact, verified a distributed version of Kruskal's algorithm.) In Gallager's algorithm *many* fragments can, as in a distributed version of Kruskal's algorithm, be combined into larger ones asynchronously from each other. Moreover, as discussed above, two fragments may combine sometimes, too, even when their minimum-weight outgoing edges do not coincide. Consequently, in Gallager's algorithm far more nondeterminism, i.e., more different interleavings, has been introduced than in those other algorithms.

The additional amount of nondeterminism, on the other hand, obviously complicates the reasoning about Gallager's algorithm, because of the vast number of generated computation sequences. Consequently, for any correctness proof of this algorithm some particular strategy must be adopted in order to obtain a transparent proof. Our strategy is the following one:

(A) First design, starting from the program in figure 1, distributed algorithms which determine the minimum-weight outgoing edge of each of the fragments constructed so far. This part of the strategy corresponds to refining the statement labeled (1) in the program in figure 1. (How to accomplish such a refinement has been described by Back [B88] and by Chandy and Misra [CM88].)

   Part (A) which deals with finding the minimum-weight outgoing edge of a fragment ($V'$, $E'$) can be split up into finding such an edge in case

   (A1) $\mid V' \mid = 1$ holds, i.e., ($V'$, $E'$) is a trivial fragment, and

   (A2) $\mid V' \mid > 1$ holds, i.e., ($V'$, $E'$) consists of at least two nodes.

   Formally, this case-distinction can be achieved by a *case-introduction* [P89]. The intuition behind this case-distinction is the following: A fragment consisting of a single node can determine its minimum-weight outgoing edge by a simple table look-up when each node has a local table assigning weights to its adjacent edges; for fragments consisting of more than one node the nodes in this fragment must, in any distributed implementation, cooperate by means of messages in order to determine the fragment's minimum-weight outgoing edge.

(B) Then design distributed algorithms in order to combine two fragments into a larger one. This part of the strategy corresponds to refining the statement labeled (2) in the program in figure 1.

   – Part (B) naturally splits up into two cases:

   (B1) One for combining two fragments which are at the same level and which have an identical minimum-weight outgoing edge, and

   (B2) one for combining a low-level fragments with a high-level one.

(C) Finally, combine the algorithms found in (A) and (B) above in order to obtain one algorithm which is the distributed version of the algorithm described in figure 1. These combinations are accomplished by applying the principle discussed in section 3.3 a finite number of times.

The distributed version of Gallager's algorithm can now be described in terms of logical tasks, as if they are performed sequentially, by refining A1, A2, and B1 even further. Task 1 describes the refinement of B1 when case A1 holds. The task 2, 3, 4, and 5 describe the refinement of B1 when case A2 holds. (How to incorporate possibility B2 is discussed in section 6.7. Incorporating the latter possibility has the effect that the sequentially performed tasks may be disturbed temporarily. As shown in section 6.7, these disturbances do not affect the reasoning about these tasks.)

*Task 1*: when a node starts participating in the algorithm it determines its minimum-weight outgoing edge (as described in A1 above) and sends a *Connect*-message along this edge. This message serves as a request from the node to combine with the fragment at the other end of this edge. (A node which receives this *Connect*-message also participates in the same task, cf section 3.) Node $i$ in $V$ participates in this task when executing the program segment labeled $A_i$ in figure 2.

Thereafter the following tasks are performed repeatedly:

*Task 2*: if two fragments have determined that they are at the same level $L$ and that they have the same minimum-weight outgoing edge, then they are combined, as described in theorem 4.2, into a larger one at level $L+1$. Node $i$ in such a fragment participates in this task when it executes the program segment labeled $B_i$ in figure 2.

*Task 3*: the weight of the minimum-weight outgoing edge of the newly formed fragment is determined. If no such edge exists, the algorithm terminates. Node $i$ participates in this task when it executes the program segment labeled $C_i$ in figure 2.

*Task 4*: if the minimum-weight outgoing edge of the newly formed fragment exists, then the node in this fragment adjacent to this edge is notified. The reason for doing so is explained in *Task 5* below. Node $i$ participates in this task when it executes the program segment labeled $D_i$ in figure 2.

*Task 5*: the node that has been notified that it is adjacent to the minimum-weight outgoing edge (cf. *Task 4* above) sends a *Connect*-message along this edge. (As described above, this message serves as a request from the fragment to combine with the fragment at the other end of this edge.) Node $i$ participates in this task when it executes the program segments labeled $E_i^1$ or $E_i^2$ in figure 2.

Note that their exist actions $a$ in the program described in figure 2, which can be executed by node $i$, that belong to program segments labeled $A_i$ and to program segments labeled by $E_i^2$. If node $i$ belongs to a trivial fragment, then such actions $a$ are considered to be part of the segment labeled $A_i$; otherwise, i.e., if node $i$ belongs to a non-trivial fragment, then these actions $a$ are considered to be part of the segment labeled $E_i^2$.

The program shown in figure 2 below will be explained and analyzed in the sections 6.1 through 6.7. The labeled boxes correspond to the program segments referred to in the description of the tasks above. We have used Gallager, Humblet, and Spira's notation [GHS83]. In [SR89b] we have discussed how a program represented by a list of responses as below can be transformed into our own notation for representing algorithms.

### 4.3 Outline of the correctness proof

In section 5 we formally specify by means of preconditions $p_i$ and postconditions $q_i$ ($i \in V$) what we mean by correctness of Gallager's algorithm. Then in section 6 we show that Gallager's program satisfies this specification. The proof is structured according to the above description of Gallager's algorithm in terms of tasks (cf. section 4.2).

We first analyze in the section 6.1 through 6.5, the programs associated with the tasks 1 through 5.

It is argued in section 6.6 that the programs above can be combined according to the proof principle described in section 3.3 because all its verification conditions are satisfied.

At the last stage of our correctness proof we incorporate the possibility that nodes in some fragment can be disturbed temporarily in the performance of their tasks by actions of nodes outside this fragment. (This includes the combinations of low-level fragments with high-level ones.) This is the subject of section 6.7. It is shown that the reasoning about the tasks described above is not invalidated, since interference-freedom of specifications can be shown. (For this reason the invariants and the termination conditions have been carried along in the specifications.)

```
(1)   reponse to spontaneous awakening (can only occur if the node is in the sleeping state)
      execute procedure wake-up
(2)   procedure wake-up
      begin
        let e be adjacent edge of minimum-weight;
        findcount :=0; se(e) :=branch; ln :=0; sn :=found; send Connect(0) on edge e            A_i
      end
(3)   reponse to receipt of Connect(l) on edge e
      begin                                                                             E_i^2
        if sn=sleeping then execute procedure wake-up fi;
        if l<ln
        then if se(e)=branch                                                                    B_i
             then fn :=w(e); ln :=ln+1; inbranch :=e; sn :=find;
                  for all edges e' ≠ e such that se(e')=branch
                  do send Initiate(ln, fn, sn) on e'; findcount :=findcount+1 od;
                  best-edge :=nil; best-wt :=∞; execute procedure test
             else place received message on end of queue
             fi
        fi
      end
(4)   reponse to receipt of Initiate(l, f, s) on edge e
      begin
        ln :=l; fn :=f; sn :=s; inbranch :=e;
        for all e' ≠ e such that se(e')=branch
        do send Initiate(ln, fn, sn) on e'; findcount :=findcount+1 od;
        best-edge :=nil; best-wt :=∞; execute procedure test
      end
(5)   procedure test
      if there are adjacent edges in the state basic
      then test-edge :=minimum-weight adjacent edge in state basic;
           send Test(ln, fn) on test-edge
      else test-edge :=nil; execute procedure report
      fi
(6)   reponse to receipt of Test(l, f) on edge e
      begin
        if sn=sleeping then execute procedure wake-up fi;
        if ln<l
        then place received message on end of queue                                            C_i
        else if fn≠f
             then send Accept on edge e
             else se(e) :=rejected;
                  if test-edge ≠ e then send Reject on edge e else execute procedure test fi
             fi
        fi
      end
(7)   reponse to receipt of Accept on edge e
      begin
        test-edge :=nil;
        if w(e)<best-wt then best-edge :=e; best-wt :=w(e) fi; execute procedure report
      end
(8)   reponse to receipt of Reject on edge e
      begin se(e) :=rejected; execute procedure test end
(9)   procedure report
      if findcount=0 and test-edge=nil then sn :=found; send Report(best-wt) on inbranch fi
(10)  reponse to receipt of Report(w) on edge e
      if e ≠inbranch
      then findcount :=findcount−1;
           if w<best-wt then best-edge :=e; best-wt :=w fi; execute procedure report
      else if sn=find
           then place received message on end of queue
           else if w=best-wt
                then halt
                else if w>best-wt then execute procedure change-root fi
                fi
           fi
      fi                                                                                        D_i
(11)  response to receipt of Change-Root
      execute procedure change-root
(12)  procedure change-root
      if se(best-wt)=branch
      then send Change-Root on best-edge
      else se(best-edge) :=branch; send Connect(ln) on best-edge                                E_i^1
      fi
```

Figure 2. The loop executed by node $i$ ($i \in V$) in a distributed version of Gallager's algorithm only Task 1 when through Task 5 are taken into account. (All variables occurring in this loop are assumed to be subscripted by $i$.)

## 5 Formal specification

In this section we formally state the specification that Gallager's program should satisfy. This specification consists of a precondition and a postcondition. In the next section it is shown that Gallager's program indeed satisfies this specification.

Let $(V, E, w)$ denote a weighted graph as in section 4. Let $T$ denote this graph's minimum-weight spanning tree. Let $S$ denote Gallager's program (cf. figure 3 in section 6). Since $S$ is a distributed program, each node maintains its own variables to perform its part of $S$. Node $i$'s variables, for $i$ in $V$, which play a role in the initial specification are the following: $sn_i$ and $se(e)$ for $e \in E_i$. Variable $sn_i$ denotes node $i$'s *node-status*; Variable $se_i(e)$ denotes the *edge-status* of edge $e$ from node $i$'s point of view. The values which these variables can take are next described and explained.

Variable $sn_i$ $(i \in V)$ can take the values

- *sleeping*, if it has not participated in the algorithm yet,
- *find*, if the node is participating in its own fragment's search for determining the minimum-weight outgoing edge (in section 6.3, it will be made more precise what "participating" in this context means), and
- *found*, in all other cases.

Initially each node in $V$ will be in the *sleeping*-state, i.e., initially no node participates in the algorithm.

Variable $se_i(e)$ $(i \in V, e \in E_i)$ can take the values

- *branch*, if the node has determined that the edge occurs in $T$,
- *rejected*, if the node has determined that the edge does not occur in $T$, or as
- *basic*, in all other cases, i.e., if the node has not yet determined whether the edge occurs in $T$.

Initially each node has marked all its adjacent edges as *basic*, of course, i.e., initially $\forall i \in V. \forall e \in E_i. se_i(e) = basic$ holds.

Each node $i$ in $V$ maintains its own message queue, $queue_i$. This queue is used to buffer received messages together with an identification of the edge along which these messages have been received. If a node's queue is non-empty, then its front element may be removed from its queue and either processed or, as we will see, placed at the end of the queue, waiting for other events to occur. For each node, the queue's capacity is assumed to be large enough to buffer all the node's unprocessed messages. It is not difficult to derive a maximum size such that each queue is able to buffer these messages. This is not the subject of this paper, however. Initially, for all nodes $i$ in $V$, $queue_i$ is empty. Denoting by $<>$ the empty queue, we thus require that initially $\forall i \in V.queue_i = <>$ is satisfied. Finally, we require that initially no edge contains any messages, i.e., $\forall i \in V.\forall e \in E_i.contents_i(e) = <>$ holds initially where $contents_i(e)$ denotes $e$'s contents of messages incoming on node $i$ ($i \in V$, $e \in E_i$).

Thus, we have the following *precondition* $p_i$ for each node $i$:
$sn_i = sleeping \wedge \forall e \in E_i.se_i(e) = basic \wedge queue_i = <> \wedge \forall e \in E_i.contents_i(e) = <>$.

Upon completion of the algorithm all messages queues and all channels must be empty, of course. In addition, the minimum-weight spanning tree must have been constructed. This implies that each node has actually participated in the algorithm and that it is not involved in any fragment's search for the minimum-weight outgoing edge, i.e., in the final state for all nodes $i$, $sn_i = found$ holds. Consequently, we must prove that upon termination of the algorithm the following holds:

$$\forall i \in V.queue_i = <> \;\wedge\; \forall i \in V.\forall e \in E_i.contents_i(e) = <> \;\wedge$$
$$\wedge\; \forall i \in V.sn_i = found \wedge (V, \bigcup_{i \in V} \{e \in E_i \mid se_i(e) = branch\}) = T.$$

We can, however, be more detailed about the postcondition. Observe that if $e \in E_{i,j}$ and $se_i(e) = branch$ hold, then this expresses that $e$ is an edge in $T$. Since $T$ is an *undirected* tree, it follows that $se_j(e) = branch$ must hold, too, i.e., if an edge is in $T$, then this edge is in the *branch*-state from the viewpoint of *both* its adjacent nodes when the algorithm terminates. Also observe that in the final state each node should have determined whether an adjacent edge occurs in $T$. As a consequence, $se_i(e) \neq basic$ is required to hold upon completion of the algorithm for all nodes $i$ and for all edges $e \in E_i$.

Altogether, the following *postcondition* $q$ is required:

$$\forall i \in V.queue_i = <> \;\wedge\; \forall i \in V.\forall e \in E_i.contents_i(e) = <> \;\wedge$$
$$\wedge\; \forall i \in V.sn_i = found \wedge (V, \bigcup_{i \in V} \{e \in E_i \mid se_i(e) = branch\}) = T \;\wedge$$
$$\wedge\; \forall i \in V.\forall e \in E_i.se_i(e) \neq basic \wedge \forall i, j \in V.\forall e \in E_{i,j}.se_i(e) = se_j(e).$$

The discussion above leads to requiring that the program $S$ should satisfy the following specification: $[p]S[q]$ holds, where $p$ is the conjunction of all the nodes' preconditions $p_i$ described above and where the postcondition $q$ is as above. Here $[p]S[q]$ means: if $S$ is executed in a state satisfying $p$, then $S$ always terminates in a state satisfying $q$ (total correctness). Observe that the above specification can be easily satisfied when the network consists of one node only. Consequently, in the remainder of this paper we assume that $|V|\geq 2$ holds (the network consists of at least two nodes). In addition, it is assumed that the network contains no self-loops, i.e., for all node $i$ in $V$, $E_{i,i}=\emptyset$. The reason for imposing this restriction is that the program in [GHS83] describing Gallager's algorithm does not necessarily construct $T$ when the network contains self-loops. (This is shown at the end of section 6.)

# 6 Gallager's algorithm

In this section it is shown that Gallager's program (cf. figure 3 at the end of this section) meets its specification. This specification has been formulated in section 5. As argued in section 4 expanding groups of nodes will repeatedly perform a certain tasks. For a single node which forms a fragment of its own this task consists of finding its minimum-weight outgoing adjacent edge and sending a *Connect*-message along this edge (cf. section 4.3). In section 6.1 it is shown how this task can be solved. The task of combining two fragments, the task of determining the weight of the minimum-weight outgoing edge, if any, the task of notifying the node in the enlarged fragment that it is adjacent to the fragment's minimum-weight outgoing edge, and the task of sending a *Connect*-message along such an edge performed by a collection of more than two nodes are analyzed in the subsections 6.2 through 6.5. The tasks are combined by repeatedly applying our principle (see section 3.3). This is the subject of section 6.6. In section 6.7 the combination of low-level fragments and high-level ones are analyzed.

## 6.1 The start of execution

In this subsection we analyze the distributed program which solves task 1 (cf. section 4) of determining a node's minimum-weight outgoing edge, when it forms a fragment of its own. A node starts participating in the algorithm when one of the following occurs:

- it responds to some command from a high-level procedure to initiate the algorithm (an "external trigger"), or
- it receives the first (algorithm-)message transmitted by some node in the graph (an "internal trigger").

A node can respond only to some command from a high-level procedure to initiate the algorithm if it is in the *sleeping* state. Since the structure of such a procedure is of no interest for the algorithm, we ignore such procedures. Instead, a node in the graph can initiate the algorithm according to its local information, that is, if it is in the *sleeping* state, by "awakening spontaneously". Many nodes in the network can awaken spontaneously, asynchronously from each other, and initiate the algorithm. We require, however, that a node can awaken spontaneously only if it is in the *sleeping* state.

When a node starts participating in the algorithm according to one of the two above-mentioned possibilities it determines its minimum-weight adjacent, hence outgoing, edge, marks this edge as a *branch*, and goes into the *found* state. It then transmits a *Connect*-message along the edge marked as *branch*. The node (at the other end of this edge) that receives this message will participate in this task, too. We consider here the program $S_1$ defined below.

**Definition 6.1** Program $S_1$, which solves the task considerd here, is the parallel composition of consisting of the program segments labeled by $A_i$ in figure 2 where $i$ is an element of the smallest set of nodes $V'$ such that

- at least one node has "awakened spontaneously" is in this set, and
- for all nodes $j$ in this set, if $j$'s minimum-weight outgoing edge is adjacent to node $\ell$, then $\ell$ is in this set, too (, because node $\ell$ will receive a *Connect*-message from node $j$). ■

This concludes the description of the first task in which a trivial fragment will participate.

In figure 2 node $i$'s actions associated with this task have been labeled by $A_i$. The variable $sn_i$ denotes node $i$'s (node-)status; $ln_i$ denotes the level of node $i$'s fragment as far as "known" to $i$; $se_i(e)$ records the edge-status of edge $e$ adjacent to node $i$. The initial value of the variable $ln_i$ is irrelevant. Note that each node $i$ also maintains a variable $findcount_i$. This variable, whose initial value is irrelevant, too, could have been omitted at this stage. Its significance will become clear when reasoning about another task (see section 6.3).

For the program $S_1$ defined above (see definition 6.1) the following holds: (recall that $V'$ denotes the set of all nodes that participate in the task considered here)

**Lemma 6.1** Assume that the precondition $p \equiv \bigwedge_{i \in V} p_i$ holds,

where $p_i \doteq sn_i{=}sleeping \wedge \forall e \in E_i.se_i(e){=}basic \wedge queue_i{=}<> \wedge \forall e \in E_i.contents_i(e){=}<>$ (see section 5). I.e., for all nodes $i$ in $V$ and for all edges $e$ in $E_i$, $sn_i$=*sleeping*, $se_i(e)$=*basic*, and all message queues and all edges are empty are satisfied initially. Let $i$ be some node in the set $V'$.

(a) If node $i$ executes the procedure wake-up, then in particular $sn_i=sleeping \wedge \forall e \in E_i.sn_i=basic$ holds as a precondition. As a postcondition for this procedure the following holds:
$sn_i=found \wedge findcount_i=0 \wedge \exists e \in E_i.\big(se_i(e)=branch \wedge \forall e' \in E_i.(e' \neq e \Rightarrow se_i(e')=basic)\big)$, i.e., node $i$ is in the *found* state, its variable $findcount_i$ has been assigned the value 0. In addition, except for one edge marked as *branch* all other edges adjacent to node $i$ are marked as *basic*.

(b) For all nodes $i$ in $V'$, $(\forall e \in E_i.se_i(e)=basic) \Rightarrow sn_i=sleeping$ is an invariant of the program above. Also, $sn_i \neq sleeping \Rightarrow (ln_i=0 \wedge findcount_i=0)$ is an invariant.

(c) If $sn_i \neq sleeping$ holds at a certain point during execution, then it remains so afterwards. (This implies that the procedure wake-up can be executed at most once.) If for a certain edge $e \in E_i$, $se_i(e)=branch$ holds at a certain point during execution then it remains so afterwards.

(d) For all $i \in V'$, $(sn_i=sleeping \vee sn_i=found) \wedge \forall e \in E_i.(se_i(e)=basic \vee se_i(e)=branch)$ is an invariant.

(e) If there exists some adjacent edge $e$ of node $i$ marked as *branch*, then $e$ is the minimum-weight outgoing edge of the fragment $(\{i\}, \emptyset)$.

(f) Upon completion of the program $S_1$ all edges connecting two nodes in $V'$ are empty and there exist exactly two neighboring nodes in $V'$ that have a message *Connect*(0) in their message queues. These messages have been received along their adjacent edges in the state *branch*.

(g) A node $i$ eventually completes its participation in the program above. This occurs when node $i$ transmits the message *Connect*(0) along its minimum-weight outgoing adjacent edge. (This is the termination condition $T_i$ of node $i$ for the program above.)

**Proof**

All these properties are verified straightforwardly. As an example, we show how property (e) can be established. That is, if $se_i(e)=branch$ holds for a certain node during execution of program $S_1$, then edge $e$ is the minimum-weight outgoing edge of the fragment $(\{i\}, \emptyset)$.
Initially, all node $i$'s adjacent edges are in the basic state. An edge can be marked as *branch*, only if node $i$ performs the assignment $se_i(e):=branch$ when executing the procedure wake-up. Obviously,

prior to the actual execution of this assignment edge $e$ has been selected to be the minimum-weight adjacent edge of node $i$. Since the graph contains no self-loops, property (e) clearly holds. (In fact, we have imposed the restriction that the graph contains no self-loops in order to ensure property (e). As shown in section 6.8, Gallager's algorithm does not necessary construct $T$ when this restriction is not satisfied.) ■

Hereafter we will denote the minimum-weight outgoing edge of some subgraph $G$ of $(V,\ E)$ by $minwedge(G)$. If the minimum-weight outgoing edge of $G$ does not exist, then $minwedge(G)$=nil holds, where nil denotes some fictitious edge.

## 6.2 Combining fragments at the same level with the same minimum-weight outgoing edge

In this subsection we will concentrate on the program associated with task 2, see section 4.2, which describes how two fragments $F'$ and $F''$ at the same level $L$ and with an identical minimum-weight outgoing edge are combined into a fragment at level $L$+1.

Recall that a fragment of $T$ has been defined as some non-empty subtree of $T$. This is a graph-oriented notion. Accordingly, a fragment is some static entity. Observe that fragments are enlarged when Gallager's program is executed. In order to reason formally about this program we need to define fragments (constructed so far) in terms of program-variables. This leads to the notion of a *B-fragment* of $T$ (see definition 6.3 below). Intuitively, a B-fragment of $T$ is some subgraph of $(V,\ E)$ constituting a fragment of $T$ such that each edge in the B-fragment is marked as *branch* from the viewpoint of both its adjacent nodes. Notice that if for a certain node $i$, $se_i(e)=branch$ holds, then the node $j$ at the other end of $e$ does not necessarily belong to the same B-fragment as $i$. This is the case when $se_j(e)\neq branch$ holds. This may occur, e.g., in the program associated with the first task (see section 6.1). There we could have that $se_i(e)=branch$, when $e$ is the minimum-weight adjacent edge of node $i$, while $se_j(e)=basic$ holds, if $e$ is *not* the minimum-weight adjacent edge of node $j$. This means that the property $se_i(e)=se_j(e)$ is not an invariant for the program describing Gallager's algorithm ($i,\ j \in V$, $e \in E_{i,j}$). This observation leads to the notion of a *B-graph*, defined next.

**Definition 6.2** A subgraph $(V',\ E')$ of $(V,\ E)$ is called a *B-graph* iff (i) and (ii) below are both satisfied:

(i) $(V', E')$ is connected.

(ii) $\forall i, j \in V'. \forall e \in E_{i,j}. \big(e \in E' \Leftrightarrow (se_i(e) = se_j(e) = branch)\big)$, i.e., it is a graph in which all edges are in the *branch*-state from the viewpoint of both its adjacent nodes. ■

**Lemma 6.2** Any connected subgraph of a B-graph is a B-graph itself. ■

Intuitively, if $se_i(e)=branch$ holds for some $i \in V$ and $e \in E_i$, then $e$ is an edge in $T$. This suggests defining fragments of $T$ in terms of B-graphs. In order to do so first notice that B-graphs may be empty. This is an immediate consequence of definition 6.2. This implies that a B-graph which constitutes a subtree of $T$ is not necessarily a fragment of $T$. Consequently, to define fragments in terms of B-graphs we need to refine the latter notion. To do so, observe that the earlier high-level description (see section 4.2) implies that fragments are enlarged. Therefore, if two nodes $i$ and $j$ are in the same B-graph at some point during execution of the algorithm, then they will remain remain in the same B-graph afterwards. Also, if a B-graph $(V', E') \subseteq T$ has been constructed, when performing the algorithm, then there is no need to consider any proper subgraph of $(V', E')$, cf. lemma 6.2, in order to find its minimum-weight outgoing edge, since this edge has been found earlier. Consequently, it suffices to consider *maximal* B-graphs in order to find their minimum-weight outgoing edges. This observation leads to the following definition:

**Definition 6.3** A *B-fragment* of $T$ is a maximal B-graph of $(V, E)$ constituting a subtree of $T$. ■

By definition, a B-fragment of $T$ is non-empty. It follows that any B-fragment of $T$ is a fragment of $T$. As $T$ is the unique minimum-weight spanning tree of $(V, E)$ we will use the term B-fragment as an abbreviation for the notion B-fragment of $T$. Also, the terms B-fragment and fragment will from now on be used interchangeably.

It remains to define the level of a B-fragment in terms of program-variables. Since each node $i \in V$ maintains a variable $ln_i$ to record the level of its own fragment (as far as "known" to that node), it is convenient to define this notion in terms of the variables $ln_i$. Note that for a fragment of the form $(\{i\}, \emptyset)$, $ln_i$ may be undefined when $sn_i=sleeping$ holds. We simply define the level of such a fragment to be 0. In all other cases the level of a fragment is the maximal value of the variables $ln_i$ for nodes $i$ in that fragment.

**Definition 6.4** A B-fragment $(V', E')$ is defined to be at level 0 when for all nodes $i \in V'$, $sn_i$=*sleeping* holds. In this case we refer to $(V', E')$ as a *sleeping* fragment. Otherwise, the fragment is called *non-sleeping*. The level of a non-sleeping B-fragment $(V', E')$ is defined to be $\max\{ln_i \mid i \in V'\}$. ■

**Remark:**

(i) A sleeping B-fragment is always of the form $(\{i\},\emptyset)$, i.e., it consists of one node. This is true because it will follow from our correctness proof that any node not in the *sleeping* state has executed the procedure wake-up *exactly* once and that for all nodes $i \in V$, edges $e \in E_i$, $sn_i \neq$ *sleeping* and $se_i(e)$=*branch* are invariance properties (cf. the lemmata 6.1, 6.3, 6.6, 6.9, and 6.10).

(ii) We will show that if $(V', E')$ is a non-sleeping fragment then for all nodes $i \in V'$, $ln_i$ is defined and $ln_i \geq 0$ is satisfied, cf. the lemmata 6.1, 6.3, 6.4, 6.7, 6.9, and 6.10. This implies that the level of any fragment is well-defined. ■

After this preparation we now focus on how two fragments $F'$ and $F''$ at the same level and with the same minimum-weight outgoing edge are combined into a larger fragment.

A fragment $F'$ at level $L$ that has found its minimum-weight outgoing edge, say $e$, informs the fragment at the other end of edge $e$ about its level and minimum-weight outgoing edge by sending a message *Connect*$(L)$ along $e$. Assume that the fragment $F''$, $F' \neq F''$, is adjacent to edge $e$. If $F''$ is at level $L$, too, and if $F''$ has informed the fragment $F'$ that it is at the same level and that is has the same minimum-weight outgoing edge, then $F'$ and $F''$ are combined into a fragment at level $L$+1. (If fragment $F''$ is at level $L$ and has transmitted a Connect-message along another edge than $e$, then the node of $F'$ that has received the Connect-message will delay this message, since no rule can be applied for combining $F'$ and $F''$ into a larger fragment (cf. section 6.7).

We now assume that at some point during execution of Gallager's algorithm the following holds:

**Assumption 1:**

$F'$=$(V', E')$ and $F''$=$(V'', E'')$ are two non-sleeping fragments, both at level $L$ with the same minimum-weight outgoing edge $e'$. ■

We also assume the following

**Induction hypothesis (IH):**

(a) If $F$ is some fragment at level $L' \leq L$ and $F$ transmits a *Connect*-message along edge $e$, then this *Connect*-message carries argument $L'$ and $e$=*minwedge*$(F)$ holds. Also, whenever the node of fragment $F$ adjacent to edge $e$ transmits the *Connect*-message along edge $e$, this edge is in the *branch*-state from the viewpoint of that node. In addition,

(b) Whenever a node in $F$ transmits a *Connect*-message along one of its adjacent edges, $sn_i$=*found* holds for all nodes $i$ in $F$. ■

The intuition behind IH(b) above is the following: when a fragment's minimum-weight outgoing edge has been found and when a *Connect*-message has been sent along this edge, then all nodes in the fragment have completed their contribution to the search for the minimum-weight outgoing edge of the fragment. It then follows from the interpretation of the variables $sn_i$, $i \in V'''$, that $sn_i$=*found* holds at the start of the program associated with the task considered here.

**Remark**: As we have seen in section 6.1 a zero-level fragment transmits a message *Connect*(0) on its minimum-weight outgoing edge when awakening. Also, this edge has been marked as a *branch* and the node is in the *found*-state when such a transmission occurs. This establishes the basis of induction. ■

Recall that we consider the case in which fragments $F'$ and $F''$ have been formed. Suppose that the nodes $i' \in V'$ and $i'' \in V''$ have exchanged *Connect*-messages along edge $e$. By assumption 1 and by the induction hypothesis (IH), see above, the *Connect*-messages carry argument $L$ and edge $e$ is the same as *minwedge*$(F')$ and as *minwedge*$(F'')$. It follows that $e$=$e'$ holds. From (IH) we obtain that both nodes $i'$ and $i''$ have placed the edge $e'$ in the *branch*-state. It follows that at that time a new fragment $F'''=(V''', E''')=(V' \cup V'', E' \cup E'' \cup \{e'\})$ has been formed. (Recall that we have assumed that $F'=(V', E')$ and $F''=(V'', E'')$ hold.) The edge $e'$ is called the *core* of the fragment $F'''$. This notion plays an important role in Gallager's algorithm (section 6.3). When the fragment $F'''$ has been formed a new task is being started by the nodes in $V'''$. This task consists of recording that fragment $F'''$ is at level $L$+1.

We assume that the fragments $F'$ and $F''$, just before combining into the fragment called $F'''$ satisfy property 1 below. This property states that any edge in $F'$ or $F''$ has the same (edge-)status from the viewpoint of both its adjacent nodes. Moreover, if some node in one of the fragments $F'$ or $F''$ has placed an edge in the *rejected* state, then this edge connects two nodes in the same fragment. Formally, we assume

**Property 1**: For all nodes $i, j \in V$ and for all edges $e \in E_{i,j}$,

(a) $i, j \in V' \Rightarrow se_i(e) = se_j(e)$ and

(b) $se_i(e)$=*rejected* $\Rightarrow j \in V'$ hold.

Similarly, we require that (a) and (b) hold with $V'$ and $E'$ replaced by $V''$ and $E''$ respectively. ■

Note that property 1 is satisfied if $F'$ and $F''$ are zero-level fragments which start participating in the task considered here.

How can this task be accomplished? I.e., how can the newly formed fragment $F'''$ be placed at level $L$+1? The answer is simple: the two nodes $i$ adjacent to the minimum-weight outgoing edge $e'$ of the fragments $F'$ and $F''$, from which fragment $F'''$ has been constructed, assign the value $L$+1 to their variables $ln_i$ after having exchanged the message *Connect*($L$) along edge $e'$. This is achieved by the program $S_2$ defined below.

**Definition 6.5** Define the program $S_2$ by $S_2 \equiv \|_{i \in V' \cup V''} B_i$ (cf. section 4.3). Recall that $V'$ and $V''$ denote the set of all nodes in the fragments $F'$ and $F''$ respectively. ■

Observe that in this program variables $fn_i$, $sn_i$, and $inbranch_i$ occur. The role of the variables $fn_i$ and $inbranch_i$ will be explained in section 6.3; the reason for placing the variable $sn_i$ in the *find*-state, for nodes adjacent to edge $e'$ is explained in section 6.7. W.r.t. these variables the property formulated below holds.

**Property 2**: For the fragment $F'''$ and for all nodes $i$ in $F'''$,

(a) if $ln_i$ >0 holds then $fn_i$ is defined. In particular, if $fn_i$ is defined, then its value is the weight of some edge in $F'''$, i.e., for a certain edge $e$ in $F'''$, $fn_i$=w($e$) holds,

(b) the values recorded by the variables $sn_i$ are different from *sleeping*, i.e., $sn_i$=*find* $\vee$ *found* holds. ■

Note that if a node $i$, with $ln_i$=0, enters the task considered here for the first time, then property 2 holds.

Let $i'$ be the node in $F'$ that has transmitted the message *Connect*($L$) along edge $e'$. Similarly, let $i''$ be the node in $F''$ that has transmitted the message *Connect*($L$) along edge $e'$. In order to reason about the program $S_2$ we assume that the following precondition for this program holds: all edges connecting nodes in the fragment $F'''$ are empty, the message queues of nodes in $F'''$ are empty, the nodes $i'$ and $i''$ are at the exit point of the statement "**if** $sn_i$=*sleeping* **then execute procedure** wake-up **fi**" in the segment labeled (3) in figure 2 ($i \in \{i', i''\}$), and all nodes $i$ in $V'''$ different from the nodes $i'$ and $i''$ are waiting for the receipt of some message. (Below, the last two requirements are denoted by $loc_i$=after"**if** $sn_i$=*sleeping* **then execute procedure** wake-up **fi**" for $i \in \{i', i''\}$ and by $loc_i$=at"$queue_i$" for nodes $i \in V'''-\{i', i''\}$ respectively, where the variable $loc_i$ denotes node $i$'s program counter.) Formally, we make the following

**Assumption 2**: When the program $S_2$ is executed,
$\forall i \in V'''.\forall e \in E'''.contents_i(e) = <> \wedge$
$\wedge\ \forall i \in V'''.queue_i = <> \wedge$
$\wedge \forall i \in \{i', i''\}.loc_i$=after"**if** $sn_i$=*sleeping* **then execute procedure** wake-up **fi**" $\wedge$
$\wedge \forall i \in V'''-\{i', i''\}.loc_i$=at"$queue_i$" holds in the initial state. ■

Note that assumption 2 holds when $F'$ and $F''$ are zero-level fragments which start, for the first time, participating in the task considered here.

**Lemma 6.3** Assume that assumption 1, assumption 2, property 1, property 2, and the induction hypothesis (IH) all hold. Then the following holds for program $S_2$:

(a) The assertions formulated in property 1, property 2 (a), (b) hold during execution of the program, i.e., they are invariance properties. Moreover during execution of $S_2$ no variable $se_i(e)$ is ever changed ($i \in V'''$, $e \in E_i$).

(b) Upon completion of the program, the fragment $F'''$ is at level $L$+1. More precisely, we have that upon completion of the program $ln_{i'}=ln_{i''}=L+1$, $inbranch_{i'}=inbranch_{i''}=e'$, and $\forall i, j \in V'''-\{i', i''\}.ln_i \leq L$ hold. (Recall that $i'$ and $i''$ denote the nodes in $V'''$ that have exchanged *Connect*-messages along the minimum-weight outgoing edge of $F'$ and $F''$, and that this edge is denoted by $e'$.)

(c) Upon completion of the program $sn_{i'}=sn_{i''}$=*find* and $fn_{i'}=fn_{i''}$=w($e'$) are satisfied. Moreover, all edges connecting the node $V'''$ as well as all messages queues of nodes are empty.

In addition, $\forall i \in V'''-\{i', i''\}.sn_i=found$ holds.

(d) A node $i$ adjacent to edge $e'$ completes its participation in this task iff $sn_i=find$ i.e., if $loc_i$=after "$sn_i := find$", where the assignment $sn_i := find$ occurs in the program segment $B_i$. Node $i$ not adjacent to edge $e'$ completes its participation if $loc_i$=at "$queue_i$ holds. (These are the termination conditions.) In the latter case node $i$ will not participate in $S_2$ at all. ■

## 6.3 Finding the minimum-weight outgoing edge of the fragment just formed

We next analyze the program associated with task 3 (cf. section 4.2).

After the fragment $F'''$ has been formed and after it has been placed at level $L+1$ the nodes in $F'''$ must determine the fragment's minimum-weight outgoing edge, if any. Any such edge $e$ must be in the state *basic* from the viewpoint of the node in $F'''$ which is adjacent to $e$. This is true because for any edge $e \in E_{i,j}$, $i \in V'''$, $j \in V$, we have that

$se_i(e)=branch \;\Rightarrow\; j \in V'''$ and

$se_i(e)=rejected \;\Rightarrow\; j \in V'''$ hold.

This follows from property 1, assumption 1, and the induction hypothesis (IH) above. Consequently, any outgoing edge of the fragment $F'''$ must be in the state *basic*.

Intuitively, $se_i(e)=basic$ holds for some $i \in V$, $e \in E_i$, if

- $e$ has not been investigated before by $i$, i.e., $i$ has not tested whether $e$ is an outgoing edge, or
- $e$ has been investigated before by $i$ and has been found to be an outgoing edge, but edge $e$ has not been the minimum-weight outgoing edge of node $i$'s fragment (at that time).

In order to determine its minimum-weight outgoing adjacent edge, a node could select its *minimum-weight outgoing adjacent edge in the state basic* and send a so-called *Test*-message along this edge. The node at the other end of that edge should then determine whether this edge joins two nodes in the same fragment. *The problem with this "solution" is that the decision whether an edge is an outgoing one has now been shifted to the receiver of the Test-message.*

The designers of Gallager's algorithm have proposed a very elegant solution for determining the minimum-weight outgoing adjacent edge of some node in $F'''$, if such an edge exists. This is described below. Any newly formed fragment carries a *name*. This name is supplied to each node in the fragment. The question arises, of course, how to assign names to fragments, since one has to ensure that distinct fragments have distinct names. In Gallager's algorithm the *name* of any non-trivial fragment is the *weight of its core* (cf. section 6.2 where we have described the notion of a core). The assumption that distinct edges have distinct weights will ensure that any non-trivial fragment has a *unique* name. Since $e'$ is the core of the fragment $F'''$, the weight $w(e')$ is the name of this fragment.

Now, after the fragment $F'''$ has been placed at level $L+1$ each node in the fragment is supplied with the fragment's name. In order to do so the two nodes $i'$ and $i''$ adjacent to the core start broadcasting an *Initiate*-message carrying the weight $w(e')$ as an argument to nodes on their "side" of the fragment $F'''$, i.e., node $i'$ and node $i''$ start broadcasting an Initiate-message to nodes in $F'$ and to nodes in $F''$ respectively. Except the name, the Initiate-message also carries two other arguments: the new level and the argument *find*. The significance of the level as an argument will be explained below; the significance of the argument *find* will be explained in section 6.7. Upon receipt of an Initiate-message node $i$ records the new name in its variable $fn_i$ (thus, $fn_i$ records the name of its fragment as far as "known" to node $i$) and the new level in its variable $ln_i$; Furthermore, the node is placed in the *find*-state, i.e., the variable $sn_i$ is assigned the value *find*. Then the edge along which node $i$ has received the Initiate-message is recorded in the variable $inbranch_i$. The reason for doing so will be explained below. Thereafter the Initiate-message is sent by node $i$ along all its adjacent edges in $F'''$ except the one identified by its variable $inbranch_i$. As such this broadcasting is similar to the broadcasting of information in Segall's PIF-protocol [Se83] when the graph constitutes a tree.

After a node in $F'''$ has sent the initiate message to all neighbors "downtree" in $F'''$ it starts searching for its minimum-weight outgoing adjacent edge. For this purpose, as argued above, it suffices for nodes to investigate edges in the state *basic* only. Now, if a node has no outgoing edges in the state *basic*, then it is done. (It has no outgoing edges.) Otherwise, it sends a *Test*-message on its minimum-weight adjacent edge in the state *basic*. This message carries two arguments: the fragment's (new) name and the fragment's (new) level as it has been recorded by the sender of the message.

A node receiving the *Test*-message waits until its own level (recorded in the variable $ln$) is greater than or equal to the one in the *Test*-message. (The reason for this delay is explained below.) If so, it checks whether the name of its own fragment equals the one in the *Test*-message. In case these names coincide, it sends a *Reject*-message back to the sender of the *Test*-message. This *Reject*-message serves for informing the node at the other end of the edge that the edge connects two nodes in the same fragment. If, on the other hand, the name of the node receiving the *Test*-message differs from the one in the *Test*-message, then the two nodes belong to different fragments. The receiver of the *Test*-message will, in this case, send an *Accept*-message back to the sender of the *Test*-message in order to inform this node that the edge connects two nodes in different fragments. These conventions enable nodes to determine whether edges are outgoing ones (see claim 1, claim 2, and assumption 3 below). The reason for a node receiving a *Test*-message to wait until its own level is greater than or equal to the one in the *Test*-message is the following: if a node receives a *Test*-message with a level greater than its own level, then

- it could be in the same fragment as the sender of the *Test*-message, while it has not yet received the new name and the new level, or
- it could be in another fragment than the sender of the *Test*-message (thus, with another name, if any).

Consequently, if the level of the receiver of the *Test*-message is too low, then it has no way of determining which of these cases actually occurs. This problem is solved by including the delay. In theorem 6.1 we show that this delay does not lead to a deadlock. (In the program describing Gallager's algorithm a node delays some message from being processed by replacing it at the end of the node's message queue.)

A node that has received a *Reject*-message along one of its adjacent edges places that edge in the *rejected*-state, since the edge connects two nodes in the same fragment, and continues its search for its minimum-weight outgoing adjacent edge by selecting the next possible one and sending a *Test*-message along this edge.

In some cases, a response to a *Test*-message is superfluous. The designer's of Gallager's algorithm have achieved some optimization w.r.t. the number of transmitted messages sent by nodes participating in the task considered here: if a node has transmitted a *Test*-message along, say, edge $e$ and it receives a *Test*-message with the same name and level as its own, then it simply marks the edge as *rejected*, since the nodes adjacent to this edge have the same name and, thus, belong to the same fragment, and continues its search for the minimum-weight outgoing adjacent edge immediately, without sending a *Reject*-message along this edge.

If a node has received an *Accept*-message as a response to one of its *Test*-messages, then it has found its minimum-weight outgoing adjacent edge.

After finding the minimum-weight outgoing adjacent edges, the nodes in $F'''$ must cooperate to determine the minimum-weight outgoing edge of $F'''$. At this stage the significance of the variables $inbranch_i$, for nodes $i$ in $V$, becomes clear. *Due to the variables $inbranch_i$, each node in $F'''$ is able to trace the path to the node adjacent to the core "on its side of the fragment".* This is true because each node in $F'''$ has recorded the edge along which the *Initiate*-message has been received and the *Initiate*-messages have flown from each of the nodes adjacent to the core "downtree on its side of the fragment $F'''$".

Before actually determining the minimum-weight outgoing edge of $F'''$, the weight of this edge is determined. This part of the algorithm is very similar to the reporting phase, describing that the required information has been received indeed, in Segall's PIF-protocol: each leaf in the fragment $F'''$ sends a *Report*-message "uptree". This message carries the weight of its minimum-weight outgoing adjacent edge. In case no such edge exists, this "weight" equal the fictitious weight $\infty$. An interior node waits until it has received all *Report*-messages from the nodes "downtree". Thereafter it sends a message *Report*$(W)$ "uptree", $W$ being the minimum of all the values received in the *Report*-messages and the weight of its own minimum-weight outgoing adjacent edge. Then it goes into the *found*-state, since its own contribution to its search to the minimum-weight outgoing edge of the fragment $F'''$ has been completed. This contribution of a node in $F'''$ to the task considered here thus consists of

- cooperating in supplying the nodes in $F'''$ with the new name and level of their fragment,
- finding its own minimum-weight outgoing adjacent edge, and
- reporting the minimum of the weights of the minimum-weight outgoing adjacent edge, including its own, of nodes "down-tree".

Eventually, the nodes $i'$ and $i''$ adjacent to the core will exchange the *Report*-messages. This enables these nodes to determine whether an outgoing edge of the current fragment $F'''$ exists. If so, these nodes are able to determine the weight of this edge and, also, on which side of the fragment this edge lies. Otherwise, i.e., if no outgoing edge exists, the algorithm terminates and the fragment $F'''$ is the minimum-weight spanning tree $T$ of the graph $(V, E)$ (cf. theorem 4.2(b)). This discussion concludes our description of the task considered in this subsection.

The program associated with this task consists of, for each node $i$ in $F'''$, the program segments labeled $C_i$ in figure 2. The program $\|_{i \in V'''} C_i$ does not describe the task, however. (Recall that $V'''$ denotes the set of all nodes in the fragment $F'''$.) The reason is that nodes outside the fragment $F'''$ also contribute to this task, because they may send *Accept*-messages (and not otherwise) to nodes in $F'''$ when they respond to *Test*-messages received from nodes in $F'''$. Consequently, we must also include the program segments of nodes outside $F'''$ that are activated to send *Accept*-messages.

**Definition 6.6** Let, for nodes $i$ outside the fragment $F'''$ which are connected by some edge with a certain node in $F'''$, the segments labeled (6) in $i$'s loop in figure 2, viz., "**response to receipt of Test(l, f) on edge** $e$" where $e$ is adjacent to fragment $F'''$, together with their bodies be denoted by $T_i$. Let $N(V''')$ denote the set of all those nodes outside $V'''$ which are connected by some edge with a certain node in $V'''$. The program associated with the task considered here is then described by $S_3 \equiv (\|_{i \in V'''} C_i) \| (\|_{i \in N(V''')} T_i)$. ■

In the program $S_3$ below, apart from variables already described, one can discern the following variables:

- *test-edge*$_i$, to record the edge being tested by node $i$ for outgoingness,
  *best-wt*$_i$, to record the minimum-weight of all the weights received so far from nodes "downtree" and the weight of node $i$'s own minimum-weight outgoing adjacent edge (determined so far), and
- *best-edge*$_i$, to record the edge that has supplied node $i$ with the value recorded by the variable *best-wt*$_i$.

Note that the variables $findcount_i$ are used to determine whether all *Report*-messages from node $i$ neighbors "downtree" has been received (cf. lemma 6.4(f) below).

**Lemma 6.4** Assume that assumption 1, assumption 2, property 1, property 2, and the induction hypothesis (IH) hold. Let program $S_2$'s postcondition (cf. lemma 6.3 above) be program $S_3$'s precondition. Then the following holds for program $S_3$:

(a) $\forall i \in V'''.(sn_i{=}\mathit{find} \vee sn_i{=}\mathit{found}) \wedge$
$\wedge\ \forall i \in V'''.\forall e \in E_i.(se_i(e){=}\mathit{basic} \vee se_i(e){=}\mathit{rejected} \vee se_i(e){=}\mathit{branch})$ is an invariant.

(b) For all $i \in V'''$, $i \notin \{i', i''\}$, $i$ will receive the message *Initiate*($L$+1, w($e$), *find*) exactly once and no node outside $V'''$ will ever receive this message. (Recall that $i' \in V'$ and $i'' \in V''$ are the two node adjacent to the core $e'$ of the fragment $F'''$.) Any such *Initiate*-message received by a certain node in $V'''$ has been transmitted by its father node when the fragment $F'''$ is assumed to be consisting of two fragments rooted at the nodes $i'$ and $i''$. The edge along which the *Initiate*-message is received by node $i$ is recorded by the variable $inbranch_i$.

Eventually, the following is satisfied:

$\forall i \in V'''.(ln_i{=}L{+}1 \wedge fn_i{=}w(e') \wedge sn_i{=}find)$, and taking into account the directions of edges as suggested by the variables $inbranch_i$, i.e., if $inbranch_i{=}e$ then edge $e$ is directed from node $i$ to the node at the other end of $e$, we also have that

$(V', \{inbranch_i \in E' \mid i \in V'\})$ forms a directed tree rooted at node $i'$ and

$(V'', \{inbranch_i \in E'' \mid i \in V''\})$ forms a directed tree rooted at node $i''$.

Furthermore, $inbranch_{i'} = inbranch_{i''} = e'$ is an invariant.

(c) For all nodes $i \in V'''$, $i \notin \{i', i''\}$, if node $i$ has received the *Initiate*-message along edge $e$, then $inbranch_i = e$ holds as a postcondition for the body of "**response to receipt of Initiate(l, f, s) on edge e**" and it will remain so afterwards.

(d) For all $i \in V'''$, $ln_i$ is non-decreasing.

(e) If node $i \in V'''$ transmits an *Initiate*-message along edge $e$, then $se_i(e){=}branch$ holds as a precondition for the corresponding action. It transmits such a message before it transmits any other messages associated with this task.

If node $i \in V'''$ receives an *Initiate*-message along edge $e$, then $se_i(e){=}branch$ holds as a precondition for the corresponding action.

(f) For all $i \in V'''$, at each point in any computation sequence if $findcount_i{=}n$ holds for some natural number $n$, then $n$ equals the number of *Initiate*-messages (with third argument *find*) minus the number of *Report*-messages processed by node $i$ that have been received along edges different from the one identified by $inbranch_i$.

(g) No node in $V'''$ will receive a *Connect*-message from any other node in $V'''$. ■

The proof of the above lemma is straightforward.

The most difficult part of the program $S_3$, and of Gallager's algorithm, is that part associated with the actual search of $minwedge(F''')$ on which we shall now concentrate.

According to the description of the task considered in this subsection each node in $V'''$ will, at any time, investigate at most one edge when it is searching for its minimum-weight outgoing edge. This observation leads to the notion of *an unanswered Test*-message. Intuitively, a *Test*-message is unanswered if it has been transmitted along some node's adjacent edge and the node has not yet determined whether that edge is an outgoing one.

**Definition 6.7**

(a) A node $i \in V'''$ has an *unanswered Test*-message *on edge* $e \in E_i$ iff $i$ has transmitted a *Test*-message along edge $e$ and the following holds: $se_i(e) \neq rejected$ and $i$ has not processed an *Accept*-message received along $e$ after it has transmitted this *Test*-message.

(b) Node $i \in V'''$ has an *unanswered Test*-message iff $i$ has an unanswered Test-message on some edge $e \in E_i$. ■

Obviously, if a node receives a *Reject*-message or an *Accept*-message along one of its adjacent edges, then the node has an unanswered *Test*-message on this edge.

We claim that when a node in $F'''$ starts participating in the task described by the program $S_3$ it has no unanswered *Test*-messages. This holds because of the following:

- When a node in $F'''$ participates in the tasks described in the subsections 6.1 and 6.2 it does not send any *Test*-messages. During execution of the programs $S_4$ and $S_5$ which will be introduced in the next two subsection no *Test*-messages will ever be sent by any node $i$ in $F'''$. (This is obvious from $S_4$'s and $S_5$'s program texts.)
- When a node starts participation in the task described in this subsection for the first time, that is, after a fragment consisting of a single node has been combined with another fragment as described in section 6.2 for the first time, it has no unanswered *Test*-messages.
- When a node has completed its participation in the task described in this subsection it has no unanswered *Test*-messages (cf. lemma 6.8(a) below).

As a consequence, the following lemma is true:

**Lemma 6.5** For all nodes $i \in V'''$,

(a) At any time $i$ has at most one unanswered *Test*-message.

(b) If $i$ has some unanswered *Test*-message on edge $e$ ($e \in E_i$), then *test-edge*$_i$=$e$ holds.

**Proof**

Both (a) and (b) are proved by an inductive argument.

From the discussion above it follows that, in order to prove the lemma, it suffices to show that the properties (a) and (b) are satisfied for the program $S_3$. From same discussion it follows that (a) and (b) hold in the initial state of the program $S_3$.

Now suppose that (a) and (b) hold up to a certain point in a computation of $S_3$ (the induction hypothesis).

(a) If node $i$ has some unanswered *Test*-message and transmits another *Test*-message thereafter, then node $i$

  (i) differs from the nodes $i'$ and $i''$ and it responses to an *Initiate*-message, i.e., it executes the program segment labeled (5) in figure 2,

  (ii) is either $i'$ or $i''$ and it executes the program segment labeled (3) belonging to the part labeled $C_i$ in figure 2,

  (iii) responses to some *Test*-message received along edge $e$ where *test-edge*$_i$=$e$ holds, (cf. the program segment labeled (6) in figure 2), or it

  (iv) responses to an *Reject*-message, i.e., it executes the program segment labeled (8) in figure 2.

Case (i) cannot occur, since this implies that a *Test*-message has been sent by node $i$ before it has transmitted an *Initiate*-message, which contradicts lemma 6.4(e), or it implies that node $i$ receives more than two *Initiate*-messages during execution of the program $S_3$, which contradicts lemma 6.4(b).

Case (ii) cannot occur because of lemma 6.4(g).

If case (iii) occurs, then, by the induction hypothesis, node $i$'s unanswered *Test*-message has been transmitted along edge $e$. This message, thus, becomes answered, i.e., not unanswered, when it processes the other *Test*-message. Therefore, when node $i$ transmits the latter *Test*-message it has no unanswered *Test*-messages. By the same argument it can be shown that the lemma remains true when case (iv) above occurs.

(b) The proof should now be obvious. ■

Using lemma 6.5, it is straightforward to prove that the following holds when the program $S_3$ is executed:

**Lemma 6.6** For all nodes $i \in V'''$ and edges $e \in E_i$ (thus, $e \neq$nil holds),

(a) If $i$ has an unanswered *Test*-message on edge $e$, then $sn_i$=*find* holds.

(b) A *Test*-message can be transmitted by $i$ only *after* it has transmitted an *Initiate*-message (with third argument *find*), and whenever $i$ transmits a *Test*-message $sn_i$=*find* holds.

(c) If $i$ receives an *Accept*-message on edge $e$, then $sn_i$=*find* $\wedge$ $se_i(e)$=*basic* $\wedge$ *test-edge*$_i$=$e$ holds. If $i$ executes (an occurrence of) the assignment $se_i(e)$ :=*rejected*, then $se_i(e)$=*basic* holds as a precondition.

(d) If $i$ transmits a *Test*-message along edge $e$, then $e$ is the minimum-weight adjacent edge of $i$ in the state *basic*, and $i$ will never receive two or more messages of the following type along this edge while performing the program $S_3$: an *Accept*-, a *Reject*-, or a *Test*-message with its own name as an argument.

(e) Once $e$ is in the *branch*-state from node $i$'s point of view, then it remains so afterwards.
During execution of the program $S_3$ no edge is placed in the *branch*-state.
Once $e$ is in the *rejected*-state from node $i$'s point of view, then it remains so afterwards. ■

Since the weight of the core is chosen as the name of any non-trivial fragment, we also have the following lemma, whose proof is obvious.

**Lemma 6.7** For all nodes $i \in V'''$,

(a) If $i$ receives a message *Initiate*($l$, $f$, $s$), then $fn_i \neq f$ holds as a precondition. Here, we assume that if $fn_i$ does not have a defined value, then it differs from any defined value.

(b) The variable $fn_i$ (for node $i$ different from $i'$ and $i''$) can change only, possibly from an undefined value to a defined one) after node $i$ has received an *Initiate*-message.

(c) If $i$ receives a message *Initiate*($l$, $f$, $s$), then $ln_i < l$ holds as a precondition. ■

It also follows that when two nodes are combined into a larger one always a *new* name is chosen. This is an immediate consequence of the fact that when $\mathrm{w}(e)$ is chosen to be the name of a fragment, $e$ is an edge of that fragment while before that moment $e$ has not been in that fragment. Consequently, it follows from lemma 6.7 that any name occurs at most at one level.

Next, consider the case that a certain node $i$ in $V'''$ has an unanswered *Test*-message on edge $e$. This implies that $i$ has transmitted a *Test*-message along edge $e$ and that it has not processed an *Accept*-, a *Reject*-, or a *Test*-message with its own name (hence, with its own level) received along edge $e$ afterwards. From lemma 6.5(b), it follows that $\mathit{test\text{-}edge}_i{=}e$ holds. Now, either (A), (B), (C), or (D) below occurs:

(A) $i$ deadlocks. That is, the *Test*-message remains unanswered. As a consequence, $\mathit{test\text{-}edge}_i{=}e$ continuously holds afterwards.

(B) $i$ will receive an *Accept*-message along edge $e$, say from node $j$.

**Claim 1:** $j \notin V'''$ holds.

**Proof:** The proof is by contradiction. Suppose that $j \in V'''$ holds. When node $j$ transmits an *Accept*-message along edge $e$, then $ln_j \geq ln_i = L{+}1$ and $fn_j \neq fn_i$ hold. Since $j \in V'''$, $ln_j \leq L{+}1$ holds, too, when executing the program $S_3$. Whence, $ln_i = ln_j$ holds. Consequently, we obtain that $fn_i = fn_j$ is satisfied (, otherwise, node $j$, in the same fragment as node $i$, would have received the same level, but yet another name than $j$; contradiction). This contradicts the assumption that $fn_i \neq fn_j$ holds when node $j$ has transmitted the *Accept*-message to node $i$.

(C) $i$ will receive a *Reject*-message along edge $e$, say from node $j$.

**Claim 2:** $j \in V'''$ holds.

**Proof:** When node $j$ transmits the *Reject*-message along edge $e$, $ln_j \geq ln_i = L{+}1$ and $fn_j = fn_i$ hold. Since $ln_i = ln_j$ holds at that time, too, it follows that the nodes $i$ and $j$ belong to the same fragment. (Recall that no node outside the fragment $F'''$ will ever receive the name $\mathrm{w}(e')$, cf. lemma 6.4).

(D) $i$ will receive a *Test*-message along edge $e$, say from node $j$, carrying the same name and level as its own.

In this case $j \in V'''$ holds. The proof is similar to the one given in (C) above.

From these case-distinction and from $S_3$'s program text, we can now conclude that eventually one of the following is satisfied for node $i \in V'''$:

(a) $i$ deadlocks. I.e., from a certain point in the computation of the program $S_3$ *test-edge*$_i$=$e$ continuously holds for a certain edge $e \in E_i$.

(b) *test-edge*$_i$=nil and $i$ has received an *Accept*-message along edge $e$, $e \in E_i$. This implies that edge $e$ is node $i$'s minimum-weight outgoing adjacent edge.

(c) *test-edge*$_i$=nil $\wedge$ $\forall e \in E_i.se(e) \neq$*basic* holds. This implies that node $i$'s has no outgoing edges.

At this stage we cannot prove that the first possibility, i.e., (a) above, will never occur. That is, we cannot conclude now that eventually each node in $V'''$ will eventually determine its minimum-weight outgoing adjacent edge (, if any). In order to do so, we have to incorporate that low-level fragments which attempt to combine with high-level ones are immediately "absorbed" by these high-level fragments. In theorem 6.1, we will show that Gallager's algorithm is deadlock-free.

At this stage we make the following assumption:

**Assumption 3**: Eventually, for all nodes $i \in V'''$, either (b) or (c) above will occur. ■

Observe that this assumption implies that eventually node $i$ will find its minimum-weight outgoing edge, provided that this edge exists.

Nodes in $V'''$ that have determined their minimum-weight outgoing adjacent edge must cooperate to determine the weight of their fragment's minimum-weight outgoing edge. This is the subject of the following lemma.

**Lemma 6.8** For the program $S_3$ the following is satisfied:

(a) Each node $i \in V'''$ will transmit exactly one *Report*-message. When this occurs $i$ has no unanswered *Test*-messages.

(b) A node $i \in V'''$ transmits the *Report*-message along the edge identified by its variable *inbranch*$_i$. Consequently, any *Report*-message is sent along an edge in the *branch*-state (cf. lemma 6.4(b,c) and lemma 6.6(e)).

(c) If node $i \in V'''$ transmits the *Report*-message then it has received a *Report*-message along each of its adjacent edges in the *branch*-state except for the one identified by its variable *inbranch*$_i$.

Observe that when a node in $V''' - \{i', i''\}$ transmits a *Report*-message along one of its adjacent edges, it has received an *Initiate*-message along that edge earlier. Due to this observation, to the property formulated in (c) above, and to the fact that no variable *inbranch*$_j$ of nodes $j$ "downtree" in $F'''$ can change after node $i$ has transmitted a *Report*-message we can consider directed subtrees of $F^{',i}$ and $F^{'',i}$ at any point of $S_3$'s execution when node $i$ in $V^{',i}$ and $V^{',i}$, respectively, transmits a *Report*-message. Define for node $i \in V'$ the directed tree $F^{',i}$, rooted at $i$ (taking into account directions suggested by the variables $inbranch_\ell$ for node $\ell$ in $V'$, cf. lemma 6.4(b)), by

$F^{',i}=(V^{',i}, E^{',i})$ where the following is satisfied:

$$V^{',i} = \begin{cases} \{i\}, \text{ if } \neg\exists \ell \in V'.\ell \neq i \wedge inbranch_\ell \in E'_i \\ \{i\} \cup \bigcup\{\ell \in V' \mid \ell \neq i \wedge inbranch_\ell \in E'_i\} \cup \bigcup\{j \in V^{',\ell} \mid \ell \neq i \wedge inbranch_\ell \in E'_i\}, \text{ otherwise} \end{cases}$$

and

$$E^{',i} = \begin{cases} \emptyset, \text{ if } \neg\exists \ell \in V'.\ell \neq i \wedge inbranch_\ell \in E'_i \\ \{inbranch_\ell \mid \ell \in V^{',i} \wedge \exists j \in V^{',i}.inbranch_j \in E_{i,\ell}\}, \text{ otherwise.} \end{cases}$$

The directed tree $F^{'',i}$ rooted at $i \in V''$ is defined in the same way. We then have the following property:

(d) For all nodes $i \in V'$, if $i$ transmits a *Report*-message with argument $W$ then $sn_i$=*found* continuously holds afterwards and $W$ equals the minimum of all weights of edges $e$ such that $e$ is an outgoing edge of the fragment $F'''$ and $e$ is adjacent to some node in the tree $F^{',i}$. Here, $W = \infty$ iff no such edge exists. The same property holds, of course, also for nodes $i \in V''$ with $F^{',i}$ replaced by $F^{'',i}$.

(e) Eventually, for all nodes $i$ in $V'''$, $findcount_i$=0 continuously holds (again). $findcount_i$=0 can only hold if $sn_i$=*found* is satisfied. Eventually, the nodes $i'$ and $i''$ will exchange a *Report*-message along the core $c'$.

(f) During execution of the program $S_3$, the following property invariantly holds for all nodes $i \in V'''$: either *best-wt*$_i$ has an undefined value, or *best-wt*$_i$ has a defined value and

$(best\text{-}wt_i{=}\infty \Rightarrow best\text{-}edge_i{=}\text{nil}) \wedge$

$\wedge\Big(best\text{-}wt_i < \infty \Rightarrow \exists e \in E_i.\big(best\text{-}edge_i{=}e \wedge (\, se_i(e){=}branch \vee se_i(e){=}basic)\,\big)\Big).$

(g) Eventually, *best-wt*$_i$ has a defined value and the value for the variable *best-wt*$_i$ has been supplied along the edge identified by the variable *best-edge*$_i$ ($i \in V'''$).

(h) When node $i$ transmits a *Report*-message, then this message carries *best-wt*$_i$ as an argument.

(i) $i'$ and $i''$ are the only nodes $i$ in $V'''$ that will receive a *Report*-message along the edges identified by the variable *inbranch*$_i$.

If in the final state of any execution of the program $S_3$, *best-wt*$_{i'}$=*best-wt*$_{i''}$ holds then this is equivalent to *best-wt*$_{i'}$=*best-wt*$_{i''}$=$\infty$, since distinct edges have distinct weights, which implies that $F'''$ has no outgoing edges, see (d) and (h) above, which implies that $F'''=T$ holds.

It follows that if the algorithm terminates, i.e., if the nodes adjacent to fragment $F'''$'s core have executed the **halt**-statement, then the minimum-weight spanning tree $T$ has been constructed. In that case the postcondition $q$ formulated in section 5 then holds.

If the algorithm does not terminate, i.e., no **halt**-statement has been executed, then *best-wt*$_{i'} \neq$ *best-wt*$_{i''}$ holds.

(j) A node $i$ in $V'''$, $i \notin \{i', i''\}$, completes its contribution to the program $S_3$ when it transmits a *Report*-message. A node $i$, $i \in \{i', i''\}$, completes its contribution to the program $S_3$ when it has both sent and received a *Report*-message along the edge $e'$, and it has either executed the **halt**-statement or it has determined that the value of its variable *best-wt*$_i$ differs from the value received in the *Report*-message. (These are the termination conditions.) ■

It should be clear that during execution of the program $S_3$, property 2(a,b), see section 6.2, invariantly holds. Also upon termination of $S_3$, property 1 and for all $i \in V'''$, $sn_i$=*found* hold.

### 6.4 Notifying the node adjacent to the fragment's minimum-weight outgoing edge

Suppose that the algorithm has not constructed the minimum-weight spanning tree. In that case, in the final state of the program $S_3$, *best-wt*$_{i'} \neq$ *best-wt*$_{i''}$ holds. The nodes in $V'''$ should accomplish the task of notifying the node in $V'''$ that it is adjacent to fragment $F'''$'s minimum-weight outgoing edge. Assume that *best-wt*$_{i'} <$ *best-wt*$_{i''}$ holds in program $S_3$'s final state. (The other case is similar.) Clearly, *best-wt*$_{i'}$ is the weight of fragment $F'''$'s minimum-weight outgoing edge. Denote this edge by $e'''$. Due to lemma 6.8(f) and (g), the path *pt* to the node $\ell$ in $V'''$ adjacent to this edge can be traced from node $i'$ by following the edges identified by *best-edge*$_i$ for nodes $i$ along *pt*. A message *Change-Root* is sent along the edges constituting the path *pt* from until this message has arrived at node $\ell$. It remains to describe how a node along the path *pt* "knows" whether it is adjacent to edge $e'''$. This is trivial, however. If for a node $i$ along *pt*, $se_i$(*best-edge*$_i$)=*branch* holds, then the edge identified by *best-edge*$_i$ is an edge in $F'''$; otherwise, $se_i$(*best-edge*$_i$)=*basic* holds (cf. lemma 6.8(f)) and the edge identified by *best-edge*$_i$ is an outgoing one.

The program $S_4$ associated with the task considered here is defined below:

**Definition 6.8** Define $S_4 \equiv \|_{i \in V'''} D_i$ (cf. figure 2 in section 4.3). ■

**Lemma 6.9** If the program $S_4$ is executed in program $S_3$'s final state for which *best-wt*$_{i'} \neq$ *best-wt*$_{i''}$ holds, then

(a) no program variable is ever changed, and

(b) nodes $i$ different from $\ell$ on the path from the node $i'$ when *best-wt*$_{i'} <$ *best-wt*$_{i''}$ is satisfied, or from the node $i''$ when *best-wt*$_{i''} <$ *best-wt*$_{i'}$ is satisfied, to the node $\ell$ in $V'''$ adjacent to *minwedge*$(F''')$ complete their participation in $S_4$ after transmitting a message *Change-Root*. Other nodes in $V'''$ different from $\ell$ never execute any statement in the program $S_4$. Node $\ell$ completes its participation in $S_4$ after it has determined that *se*$_\ell$(*best-edge*$_\ell$) $\neq$*branch* holds. (Cf. the program segment labeled (12) in figure 2.) ■

## 6.5 Sending a Connect-message on the minimum-weight outgoing edge

After the nodes in the fragment $F'''$ have determined the weight of $F'''$'s minimum-weight outgoing edge $e'''$ and after node $\ell$ in $F'''$ adjacent to $e'''$ has been notified about this, the fragment $F'''$ attempts to combine with the fragment, say $F''''$, at the other end of $e'''$. In order to do so, node $\ell$ sends a *Connect*-message carrying $F'''$'s level, i.e., $L+1$, as its argument. Assume that $F''''=(V'''', E'''')$ holds and that node $k \in V''''$ is adjacent to edge $e'''$. Also assume that the fragment $F''''$ is at level $L+1$ and that $k$ has transmitted a *Connect*-message along edge $e'''$, too. Then the two fragments will be combined into a larger fragment $F'''''$ as described in section 6.2. For $i \in V''' \cup V''''$, node $i$ participates in the task of combining these fragments (as described above), when it executes the program segments labeled $E_i^1$ or $E_i^2$ in the figure shown in figure 2.

**Definition 6.9** Let $G_i$ denote the program segment consisting of node $i$'s program segments labeled $E_i^1$ or $E_i^2$. Define $S_5 \equiv \|_{i \in V''' \cup V''''} G_i$. ■

**Lemma 6.10** Under the aforementioned assumptions, lemma 6.3 holds for the program $S_5$ when in that lemma $F'''$, $L$, $e'$, $i'$, and $i''$ are replaced by $F'''''$, $L+1$, $e'''$, $\ell$, and $k$ respectively. ■

Observe that if node $\ell$ in the fragment $F'''$ transmits a *Connect*-message along edge $e$, then this message carries $F'''$'s level as an argument and $minwedge(F''')$ holds. Also observe that the *Connect*-message is then transmitted along an edge marked as a *branch*. From the property formulated after lemma 6.8 and from lemma 6.9, it also follows that all nodes in the fragment $F'''$ are in the *found*-state when the *Connect*-message is sent. This establishes the induction step (cf. section 6.2, where the induction hypothesis (IH) has been formulated).

## 6.6 Combining the above specifications

Above we have associated a specification to each program describing one of the subtasks (cf. section 4.3). Each specification consists of, for each node $i$ participating in the respective program, a precondition $pre_i$, a postcondition $post_i$, an invariant $I_i$, and a termination condition $T_i$. These assertions have been formulated in the lemmata 6.1 through 6.10. We now apply the principle of section 3.3 in order to obtain one algorithm that describes that from a logical point of view the five tasks are performed sequentially and repeatedly. In order to do so, observe that the programs which have been analyzed above may involve distinct set of nodes. This can be seen, e.g., with the programs $S_2$ and $S_3$. Program $S_2$ describes how two fragments $F'$ and $F''$ are combined into a larger fragment $F'''$ (see section 6.2). In this program all nodes of the fragment $F'''$ are considered. Whereas in program $S_3$, which describes how the minimum-weight outgoing edge of fragment $F'''$ is determined, apart from nodes in $F'''$ also neighboring nodes of $F'''$ are considered.

The principle below states how the set of nodes involved in a certain program can be augmented while preserving all properties of the original program.

The intuition behind this principle is as follows:

Let $\mathcal{D}=< V', \{p_i \mid i \in V'\}, Act^{\mathcal{D}} >$ be some algorithm. By assumption (see section 2), no node outside $V'$ is actually involved in $\mathcal{D}$. Let $V''$ be some set of nodes satisfying $V' \subseteq V''$. Nodes in $V'' - V'$ do not actually participate in $\mathcal{D}$ (as has been observed above). Consequently, if $p_i$ is an arbitrary state assertions of nodes $i \in V''-V'$) characterizing node $i$'s precondition and if $p_i$ does not refer to variables which can be changed by nodes different from $i$, then $p_i$ is an invariant and a termination condition for node $i$ ($i \in V'' - V'$ when the algorithm $\mathcal{D}'=< V'', \{p_i \mid i \in V'\} \cup \{p_i \mid i \in V'' - V'\}, Act^{\mathcal{D}} >$ is executed. This idea leads to the following principle:

- Let $\mathcal{D} = < V', \{p_i \mid i \in V'\}, Act^{\mathcal{D}} >$ be some algorithm.
- Let $\mathcal{D}$ **sat** $< \{I_j \mid j \in V'\}, \{T_j \mid j \in V'\}, \{q_j \mid j \in V'\} >$ hold.
- Let $V''$ be a set satisfying $V' \subseteq V'' \subseteq V$.
- Let for $j \in V'' - V'$ state assertions $p_j$ be given. Assume that none of these assertions contain any programming variables which can be changed by actions of nodes different from $j$, and that they do not contain proof variables $\rho_\ell(e)$ and $\sigma_\ell(e)$ for nodes $\ell \neq j$.
- Define for $j \in V'' - V'$, $I_j \equiv p_j$, $T_j \equiv p_j$, and $q_j \equiv p_j$.

  Then the following is satisfied for algorithm $\mathcal{D}' = < V'', \{p_i \mid i \in V' \cup V''\}, Act^{\mathcal{D}} >$:

- $\mathcal{D}'$ **sat** $< \{I_j \mid j \in V''\}, \{T_j \mid j \in V''\}, \{q_j \mid j \in V''\} >$.

The soundness of this principle is obvious.

We now combine the programs that have been analyzed in the sections 6.1 through 6.5. Each one descibes how some fragment solves a certain task. In order to do so, we may assume, as described by the principle above, that all programs involve the same set of nodes. The combination can then be achieved by means of the principle for formal sequentially phased reasoning (see section 3.3). It must therefore be shown that all verification conditions required for a sound application of the latter mentioned principle are satisfied. For each of the programs involved in the combination, we have derived invariants and termination conditions in the lemmata 6.1 through 6.10. It is straightforward to verify all the other verification conditions (cf. also section 6.7 for the case in which a *Connect*-message is received by a node too "early", i.e., if this message is received along an edge not in the *branch*-state). The complete proofs are, however, quite lengthy and do not provide us with more insight in Gallager's algorithm. Therefore, as an illustration that all verification conditions of the principle are satisfied, we concentrate on the requirement that each node can (actually) participate in one subtask at a time. We consider two cases:

(I) A node which participates in the program $S_2$ cannot participate in the program $S_5$. (These programs have been defined in the sections 6.2 and 6.5.) This holds because of the following: If node $i$ participates in program $S_2$, then $sn_i$=*sleeping* $\vee$ $ln_i$=0 holds. If node $i$ participates in program $S_5$, then $sn_i$=*found* holds and it has received a message *Change-Root*, which in turn implies that it has increased its level earlier, i.e., $ln_i$ >0 holds. It is now obvious that node $i$

cannot participate in the programs $S_2$ and $S_5$ at the same time.

(II) A node cannot participate in the program $S_2 \equiv \|_{i \in V'''} B_i$ when it is part of a fragment ($V'''$, $E'''$) at level $L$+1 (cf. section 6.2) while it is participating in the program $S_2' \equiv \|_{i \in V''''} B_i$ when it is part of a fragment ($V''''$, $E''''$) at level $L$. This follows from the following:
If node $i$ participates in the program $S_2$, then it has received a *Connect*-message with argument $L$+1 along an edge marked as a *branch*. It follows that $In_i$=$L$+1 holds when it has received this *Connect*-message. If it would at the same time participate in the program $S_2'$, then it starts participating in this program when $In_i < L$ holds (cf. section 6.2); contradiction.

## 6.7 The full version of Gallager's algorithm

We now consider Gallager's program. In this program different group of nodes perform their tasks concurrently w.r.t. another. Furthermore, a task performed by one group of nodes can be disturbed (temporarily) due to interference with the task of an other group.

At first, we describe how to combine two programs performed by two disjoint groups of nodes. Intuitively, these programs are executed completely independent of each other. A principle for combining such programs is straightforward:

- Let $\mathcal{A}$=< $\{V', \{p_i \mid i \in V'\}, Act^{\mathcal{A}}$ > and $\mathcal{B}$=< $\{V'', \{p_i \mid i \in V''\}, Act^{\mathcal{B}}$ > be algorithms.
- Assume that $V' \cap V'' = \emptyset$ holds (no node is involved in both algorithms).
- Assume that $\mathcal{A}$ **sat** < $\{I_j \mid j \in V'\}, \{T_j \mid j \in V'\}, \{q_j \mid j \in V'\}$ > and $\mathcal{B}$ **sat** < $\{I_j \mid j \in V''\}, \{T_j \mid j \in V''\}, \{q_j \mid j \in V''\}$ > hold.
- Assume that none of the assertions $p_j$, $I_j$, $T_j$, and $q_j$, $j \in V' \cup V''$, contains any programming variables of nodes different from $j$ and that they do not refer to proof variables $\rho_\ell(e)$ and $\sigma_\ell(e)$ for $\ell \neq j$.

  Then for algorithm $\mathcal{C}$=< $V \cup V'', \{p_i \mid i \in V' \cup V''\}, Act^{\mathcal{D}}$ >:
- $\mathcal{C}$ **sat** < $\{I_j \mid j \in V' \cup V''\}, \{T_j \mid j \in V' \cup V''\}, \{q_j \mid j \in V' \cup V''\}$ > holds.

We have described how programs which are executed completely independent from each other can be combined into one algorithm. Next, we consider the possibility that nodes in a fragment $F$ can be disturbed (temporarily) when they participate in one of the tasks discussed above. Consequently, we ask ourselves the question what messages nodes in $F$ can receive from nodes outside $F$ when they perform a certain task The answer of this question shows that some minor changes in the program of figure 2 have to be made and that some of the assertions derived in the previous subsections have to be weakened.

A node in a fragment $F$ can obviously receive *Accept*-, *Test*-, and *Connect*-messages (not otherwise) from nodes outside $F$.

An *Accept*-message can be send by some node $j$ outside $F$ to a certain node $i$ in $F$ only, if it has received a *Test*-message from node $i$ earlier, i.e., if node $i$ participates in the task described in section 6.3. Since responding to *Test*-messages by means of *Accept*-messages is part of that task, node $i$ is not disturbed in the performance of its task.

Now suppose that node $j$ outside the fragment $F$ sends a *Test*-message to node $i$ in $F$. *Observe that this implies that we have to incorporate in the assertions of node i associated with the tasks discussed in the sections 6.1 through 6.5 that Test-messages can be received and that they are placed at the end of node i's message queue.* This is straightforward, however. Now, if node $i$ is in the *sleeping*-state, then it be awakened by this message and it will start participating in the task described in section 6.1. Therefore assume that $i$ is not in the *sleeping*-state. Node $i$, when receiving the *Test*-message, will be disturbed in the performance of the task in which it participates. When the *Test*-message is removed from node $i$'s queue, it is either places this message back at the end of its queue (if $ln_i$'s value is less than the value of the level's argument in the *Test*-message) or it sends an *Accept*-message back to the sender of the *Test*-message. In any case, node $i$ will execute the program segment labeled (6) in figure 2. During this execution none of node $i$'s program variables are changed. Consequently, the invariant associated with the task in which it participates remain valid when it executes this segment. In addition, since this execution will always leave the program segment labeled (6) it will resume its participation in the disturbed task. Note that if node $i$ has not finished this participation when being disturbed, then this remains so afterwards during $i$'s response to the receipt of the *Test*-message; otherwise, i.e., if it has completed its participation in the task when responding to the *Test*-message, then its participation in this task remains completed afterwards (cf. also verification condition (j) of the principle in section 3.3).

The most difficult case of interference occurs when node $i$ receives a message *Connect*($L$) from some node outside its fragment $F$. Obviously, if $i$ is in the *sleeping* state, then it will be awakened and start participating in the task described in section 6.1. We therefore assume that, when node $i$ receives this message, it is in the not in the *sleeping*-state. Assume that node $j$ transmitted the *Connect*-message. At the moment of transmission $ln_j{=}L$ holds. Now,

- either $L{=}0$ holds, or

  $L > 0$ holds and node $j$ has received an *Accept*-message along edge $e$ earlier. When node $i$ transmitted this *Accept*-message $ln_j \leq ln_i$ holds. Since levels are non-decreasing (cf. lemma 6.4(d)) and node $j$'s own level cannot increase after the receipt of the *Accept*-message and before the transmission of the *Connect*-message, it follows that $ln_j \leq ln_i$ holds when node $j$ transmits the *Connect*-message.

From these two cases it follows that whenever node $i$ receives a message *Connect*($L$) and checks whether $ln_i{=}L$ holds, this test is equivalent to checking whether $\neg(ln_i < L)$ is satisfied. (In the final version of the program, see figure 3 below, this observation has been taken into account.)

Now, when node $i$ receives the message *Connect*($L$) along edge $e$, $L \leq ln_i$ (see above) and $se_i(e){=}basic$ $\vee$ $se_i(e){=}branch$ (cf. property 1 in section 6.2) both hold.
If $ln_i{=}L$ and $se_i(e){=}branch$ hold, then node $i$ proceeds as described in the sections 6.1 and 6.5.
If $ln_i{=}L$ and $se_i(e){=}basic$ hold, then the *Connect*-message is delayed. (This case is similar to delaying a *Test*-message, see above).
If, on the other hand, $L < ln_i$ is satisfied, then it follows from the induction hypothesis (IH), see section 6.2, and from lemma 6.8(c) that for all nodes $k$ in $j$'s fragment, say $F'$, $sn_k{=}found \wedge findcount_k{=}0$ holds, when node $j$ transmitted the *Connect*-message. It also follows from (IH) that edge $e$ is fragment $F'$'s minimum-weight outgoing edge. Note that upon $i$'s receipt of the *Connect*-message along edge $e$,

- $fn_i$ is defined. This is true because $ln_i > 0$ is satisfied (as a consequence of $0 \leq L < ln_i$) and property 2 (see section 6.2) holds.

- Fragment $F'$'s level equals $L{=}ln_j$, which follows from (IH).

- $se_i(e){=}basic$ holds (cf. property 1, section 6.2).

From the description of Gallager's algorithm in section 4 it follows that the fragments $F$ and $F'$ are immediately combined into a larger fragment. Therefore, upon receipt of the *Connect*-message node $i$ marks edge $e$ as a *branch*. (At that time a new fragment has been formed.) Thereafter, node $i$ supplies the nodes in the fragment $F'$ with the name and level of its own fragment (as far as "known" to $i$).

Consequently, the variables $ln_k$, $k \in V$, increase indeed. We now consider three cases which can hold when node $i$ responses to the *Connect*-message:

(a) node $i$ has not yet received fragment $F$'s new name, i.e., it has not yet received an *Initiate*-message with third argument *find*,

(b) node $i$ has received fragment $F$'s new name, but it has not yet transmitted a *Report*-message, i.e., it is participating in the task described in section 6.3, or

(c) node $i$ has received fragment $F$'s new name and it has transmitted a *Report*-message.

In case (b) above, obviously, $sn_i$=*find* holds. It will immediately transmit the message *Initiate*($ln_i$, $fn_i$, $sn_i$) such that all nodes in $F'$ will participate in the enlarged fragment's search for its minimum-weight outgoing edge. The invariants derived in section 6.3 clearly remain valid. Also the termination conditions of the nodes are not changed, i.e., interference-freedom of specifications can be proved.
In case (c) there is no need for the nodes in $F'$ to participate in the (already completed) search for $F$'s minimum-weight outgoing edge since the nodes in $F'$ will not contribute anything to this search. The reason is the following:
node $i$ has transmitted a *Report*-message by assumption. Therefore node $i$ has determined its minimum-weight outgoing adjacent edge. Consequently, $best\text{-}wt_i \leq$w($e$) then holds, since edge $e$ is one of node $i$'s outgoing edges.
**Claim**: $best\text{-}wt_i <$w($e$) holds, too.
**Proof**: The proof is by contradiction. Suppose that $best\text{-}wt_i$=w($e$) holds. This implies that node $i$ has received an *Accept*-message along edge $e$ earlier, since $e$ has provided the value for $best\text{-}wt_i$ (cf. lemma 6.8). When node $j$ transmitted this message $ln_j \geq ln_i$ holds. It follows that $ln_i$ has decreased afterwards; contradiction. ■
We obtain that, in this case, for all outgoing edges $e_1$ of fragment $F'$, w($e_1$)$\geq$w($e$)$>$ $best\text{-}wt_i$ holds.
Consequently, in the cases (a) and (c), contrary to case (b), the nodes in $F'$ should synchronize their search for the minimum-weight outgoing edge of the enlarged fragment with nodes in the fragment $F$, i.e., they should wait for this search until they have received the name and level of the enlarged fragment. The cases (a) and (c) are distinguished from case (b) by the third argument in the *Initiate*-message. If a node receives an *Initiate*-message, then it updates its variable $sn$ according to the third argument of the message. It starts searching for its minimum-weight outgoing adjacent edge only, if it is in the *find*-state (cf. section 6.3). (*Initiate*-messages with a third argument *found* propagate through

the fragment $F'$ in exactly the same way as the information in Segall's PIF-protocol is propagated. The invariants and termination conditions for this part of the algorithm are very similar to the ones defined in [SR89b].) This observation has been incorporated in the program below. Note that the assertions, as before, defined in the previous subsections have to be (slightly) weakened, since now nodes can receive *Initiate*-messages with third argument *found*, but that, again, interference-freedom of specifications can be shown.

Note that whenever some node $k$ executes (an occurrence of) the assignment $se_k(e){:=}\mathit{rejected}$ for a certain edge $e \in E_k$, $se_k(e){=}\mathit{basic}$ holds as a precondition, cf. lemma 6.6. Consequently, we can replace each such an assignment by the conditional **if** $se_k(e){=}\mathit{basic}$ **then** $se_k(e){:=}\mathit{rejected}$ **fi** without affecting any of our earlier results. This modification is, however, *necessary* in order to avoid the following (unintended) situation:

node $i$ sends a *Test*-message along edge $e$, before it receives along edge $e$ a message $Connect(L)$ with $L < ln_i$;

node $i$ receives a message $Connect(L)$ with $L < ln_i$ along edge $e$;

node $i$ places edge $e$ in the *branch*-state and sends a message $Initiate(ln_i,\ fn_i,\ sn_i)$ along $e$ (observe that $sn_i{=}\mathit{find}$ holds);

node $i$ receives a message *Reject* along $e$ and places the edge $e$ in the *rejected*-state.

Consequently, $e$ has been placed in the *rejected* state by node $i$. Edge $e$ is, however, an edge in the spanning tree $T$, because the node different from $i$ adjacent to $e$ has determined that $e$ occurs in $T$.

Taking this modification and the two observations above into account, we arrive at the program in figure 3 below. This program describes (the full version of) Gallager's algorithm.

```
The program segments (1), (2), (5), (7), (9),···, (12) are the same as the ones in figure 2.
 (3)    reponse to receipt of Connect(l) on edge e
        begin
          if sn=sleeping then execute procedure wake-up fi;
          if l<ln
          then se(e):=branch; send Initiate(ln,fn,sn) on edge e;
                 if sn=find then findcount :=findcount+1 fi
          else if se(e)=basic
               then place received message on end of queue
               else fn :=w(e); ln :=ln+1; inbranch :=e; sn :=find;
                    for all edges e' ≠ e such that se(e')=branch
                    do send Initiate(ln, fn, sn) on e'; findcount :=findcount+1 od;
                    best-edge :=nil; best-wt :=∞; execute procedure test
               fi
          fi
        end
 (4)    reponse to receipt of Initiate(l, f, s) on edge e
        begin
          ln :=l; fn :=f; sn :=s; inbranch :=e;
          for all e' ≠ e such that se(e')=branch
          do send Initiate(ln, fn, sn) on e'; if sn=find then findcount :=findcount+1 fi od;
          best-edge :=nil; best-wt :=∞; if sn=find then execute procedure test fi
        end
 (6)    reponse to receipt of Test(l, f) on edge e
        begin
          if sn=sleeping then execute procedure wake-up fi;
          if l<ln
          then place received message on end of queue
          else if fn≠f
               then send Accept on edge e
               else if se(e)=basic then se(e) := rejected fi;
                    if test-edge≠ e then send Reject on edge e else execute procedure test fi
               fi
          fi
        end
 (8)    reponse to receipt of Reject on edge e
        begin if se(e)=basic then se(e) :=rejected fi; execute procedure test end
```

Figure 3. The loop executed by node $i$ ($i \in V$). (Variables occurring in this loop are assumed to be subscripted by $i$.) The program consisting of all these loops describes Gallager's algorithm.

A principle which underlies the above kind of reasoning w.r.t. the disturbances is next formulated. For ease of exposition, we consider the case that at most one node $k$ can be disturbed in the performance of its task.

Let this task be solved by algorithm

(C1) $\mathcal{B}=< V', \{p_i^{\mathcal{B}} \mid i \in V'\}, Act^{\mathcal{B}} >$.

Since node $k$ can be disturbed in the performance in $\mathcal{B}$, $k$ may receive messages from nodes outside $V'$. Receiving and processing such messages are actions associated with another algorithm, say,

(C2) $\mathcal{C}=< \{V'', p_i^{\mathcal{C}} \mid i \in V''\}, Act^{\mathcal{C}} >$.

From the assumption that $k$ is the only node that may be disturbed (due to actions in $\mathcal{C}$), it follows that we may assume that

(C3) $V' \cap V''=\{k\}$ is satisfied.

Since $\mathcal{B}$ and $\mathcal{C}$ solve distinct tasks, we may assume that

(C4) $Act^{\mathcal{B}} \cap Act^{\mathcal{C}} = \emptyset$ (this is the case in Gallager's program indeed).

Next, suppose that

(C5) $\mathcal{B}$ **sat** $< \{I_j^{\mathcal{B}} \mid j \in V'\}, \{T_j^{\mathcal{B}} \mid j \in V'\}, \{q_j^{\mathcal{B}} \mid j \in V'\} >$ and $\mathcal{C}$ **sat** $< \{I_j^{\mathcal{C}} \mid j \in V''\}, \{T_j^{\mathcal{C}} \mid j \in V''\}, \{q_j^{\mathcal{C}} \mid j \in V''\} >$ have been proved.

(C6) Assume that no assertion subscripted by $j$ can ever be changed by actions of nodes different from node $k$ (cf. verification condition (3) in section 3).

Now at any time in $\mathcal{B}$'s computation, node $k$ must allow to be disturbed by actions occurring in $Act^{\mathcal{C}}$. This is the case if the invariant $I_k^{\mathcal{B}}$ holds whenever node $k$ starts participating in algorithm $\mathcal{C}$. In particular, this is satisfied when $p_k^{\mathcal{C}} \equiv I_k^{\mathcal{B}}$ is satisfied. When node $k$ is participating in algorithm $\mathcal{C}$, i.e., when it executes an action associated in $Act^{\mathcal{C}}$, the reasoning about algorithm $\mathcal{B}$ should remain valid.

Define, for assertions $P$ and $Q$ and for a set of actions $AC$, the assertion *Int-free*$(P, Q, AC)$ expressing that if some action $a$ is executed in a state satisfying $P \wedge Q$, then $P$ is not invalidated by $a$ (interference-freedom).

We require that for all node $j \in V'$ the following holds:

(C7) *Int-free*$(I_j^{\mathcal{B}} \wedge \neg T_j^{\mathcal{B}}, I_\ell^{\mathcal{C}} \wedge \neg T_\ell^{\mathcal{C}}, Act_\ell^{\mathcal{C}})$ and *Int-free*$(I_j^{\mathcal{B}} \wedge T_j^{\mathcal{B}}, I_\ell^{\mathcal{C}} \wedge \neg T_\ell^{\mathcal{C}}, Act_\ell^{\mathcal{C}})$ $(\ell \in V'')$.

Of course, it must also be required that the reasoning about algorithm $\mathcal{C}$ remains valid under actions of $\mathcal{B}$:

(C8) *Int-free*$(I_j^{\mathcal{C}} \wedge \neg T_j^{\mathcal{C}}, I_\ell^{\mathcal{B}} \wedge \neg T_\ell^{\mathcal{B}}, Act_\ell^{\mathcal{B}})$ and *Int-free*$(I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}}, I_\ell^{\mathcal{B}} \wedge \neg T_\ell^{\mathcal{B}}, Act_\ell^{\mathcal{B}})$ $(j \in V'', \ell \in V')$.

The kind of disturbances appearing in Gallager's program can occur only when a node participates in a certain task and it receives a message associated with another task. As remarked above, such a node must at any time be prepared to receive these kinds of messages. Internal actions and send actions of node $k$ associated with different tasks cannot be enabled simultaneously, however. (This observation holds for Gallager's program.) We, thus, require that

(C9) for each action $a \in IS_k^{\mathcal{B}}$, *disabled*$(I_k^{\mathcal{B}} \wedge \neg T_k^{\mathcal{B}} \wedge en(a), IS_k^{\mathcal{C}})$ holds for all computation sequences of $\mathcal{B}$ and similarly that

(C10) for each action $a \in IS_k^{\mathcal{C}}$, *disabled*$(I_k^{\mathcal{C}} \wedge \neg T_k^{\mathcal{C}} \wedge en(a), IS_k^{\mathcal{B}})$ holds for all computation sequences of $\mathcal{C}$ (cf. section (3) for the definitions of the sets $IS_k^{\mathcal{B}}$ and $IS_k^{\mathcal{C}}$ and for the definition of the assertion *disabled*).

Finally, we require that actions associated with algorithm $\mathcal{C}$ cannot enable nor disable actions associated with algorithm $\mathcal{C}$ and that actions associated with $\mathcal{C}$cannot enable nor disable actions associated with $\mathcal{D}$:

(C11) *Int-free*$(\neg en(a), I_k^{\mathcal{B}} \wedge \neg T_k^{\mathcal{B}}, Act_k^{\mathcal{B}})$ for all $a \in IS_k^{\mathcal{C}}$,
*Int-free*$(en(a), I_k^{\mathcal{B}} \wedge \neg T_k^{\mathcal{B}}, Act_k^{\mathcal{B}})$ for all $a \in IS_k^{\mathcal{C}}$,
*Int-free*$(\neg en(a), I_k^{\mathcal{C}} \wedge \neg T_k^{\mathcal{C}}, Act_k^{\mathcal{C}})$ for all $a \in IS_k^{\mathcal{B}}$,
*Int-free*$(en(a), I_k^{\mathcal{C}} \wedge \neg T_k^{\mathcal{C}}, Act_k^{\mathcal{C}})$ for all $a \in IS_k^{\mathcal{B}}$.

If (C1),...,(C11) are all satisfied, then we may then conclude that for the algorithm $\mathcal{D}=< V' \cup V'', \{p_i^{\mathcal{B}} \mid i \in V''\} \cup \{p_i^{\mathcal{C}} \mid i \in V'' - \{k\}\}, Act^{\mathcal{B}} \cup Act^{\mathcal{C}} >$ the following holds:

$$
\begin{aligned}
\mathcal{D}\ \mathbf{sat}\ <&\{I_j^{\mathcal{B}} \mid j \in V' - \{k\}\} \cup \{I_j^{\mathcal{C}} \mid j \in V'' - \{k\}\} \cup \{I_k^{\mathcal{B}} \vee I_k^{\mathcal{C}}\},\\
&\{I_j^{\mathcal{B}} \wedge T_j^{\mathcal{B}} \mid j \in V' - \{k\}\} \cup \{I_j^{\mathcal{C}} \wedge T_j^{\mathcal{C}} \mid j \in V'' - \{k\}\} \cup \{I_k^{\mathcal{B}} \wedge I_k^{\mathcal{C}} \wedge T_k^{\mathcal{B}} \wedge T_k^{\mathcal{C}}\},\\
&\{q_j^{\mathcal{B}} \mid j \in V' - \{k\}\} \cup \{q_j^{\mathcal{C}} \mid j \in V'' - \{k\}\} \cup \{q_k^{\mathcal{B}} \wedge q_k^{\mathcal{C}}\}>.
\end{aligned}
$$

We have the following:

**Theorem 6.1** The program $S$ described in figure 3 above meets its specification (cf. section 5).

**Proof:** From the previous lemmata and the above discussions it should be clear that the program $S$ is partially correct w.r.t. precondition $p$ and postcondition $q$, where $p$ and $q$ have been defined in section 5. In order to prove that $S$ always terminates when executed in an initial state satisfying $p$, it suffices to prove that in any non-terminal state reached during execution of $S$ some (proper) progress can be made. Consider some state which can be reached during such an execution. We may assume that in this state for all nodes $i \in V$, $sn_i \not= sleeping$ holds, since otherwise at least one node could "awake spontaneously" and, thus, progress could be made.

Let *Frag* be the set of all fragments in the considered state. Let *LFrag*$\subseteq$*Frag* be the set of all fragments which have the lowest level amongst all fragments in *Frag*. Define $F \in$*LFrag* to be a fragment with the smallest minimum-weight outgoing edge among the fragments in *LFrag*.

(a) Suppose that some node in the fragment $F$ has transmitted a *Test*-message. Because of the choice of $F$, eventually this *Test*-message will become answered (either by an *Accept*-message or by a *Reject*-message).

(b) Suppose that some node in the fragment $F$ has transmitted a *Connect*-message along a certain edge $e \in E_i$. Then this node will, again by the choice of $F$, either receive a *Connect*-message along edge $e$, or it will receive an *Initiate*-message along edge $e$. Consequently, eventually the fragment's level will increase. Again, progress will be made.

(c) In all other cases it should be clear that progress is ensured. ■

## 6.8 Some notes on Gallager's algorithm

The correctness of Gallager's algorithm heavily depends on properties of the underlying network. As we have seen in the sections 6.3 through 6.5, the possibility of identifying edges by their weights is crucial for its correctness. Another, less obvious, constraint which is essential to construct the minimum-weight spanning tree using this algorithm is that the underlying network contains no self-loops, i.e., that there are no edges $e \in E_{i,i}$ for any node $i$. This property has actually been used in lemma 6.1(e). In case the network does contain self-loops it is not ensured that Gallager's algorithm indeed finds the minimum-weight spanning tree $T$ of the network. As an example, assume that there exists some edge $e \in E_{i,i}$ for a certain node $i$ in $V$. Assume that $e$ is the minimum-weight adjacent edge of node $i$ holds, too. When $i$ awakens it will mark $e$ as a *branch*. Consequently, from node $i$'s point of view $e$ will always in the *branch* state afterwards. It follows that in such a case the algorithm cannot satisfy its specification. One can slightly relax the assumption that the graph must not contain any self-loops in order to construct $T$ using Gallager's algorithm: if a node's adjacent edge is a self-loop, then it is not the node's minimum-weight adjacent edge. It can be proved that if this condition holds, then Gallager's algorithm is correct.

Our program describing Gallager's algorithm is slightly more efficient than the program in [GHS83]. While we merely update program variables of nodes that have exchanged a *Connect*-message along some adjacent edge (see figure 3), in the program in [GHS83] the nodes $i$ adjacent to this edge, say $e$, *first* exchange a message *Initiate*($ln_i$+1, w($e$), *find*), *after* having exchanged the message *Connect*($ln_i$), and *before* they broadcast the *Initiate*-message to the other nodes in their fragment. Obviously, we have saved some transmissions of messages when compared with the program in [GHS83].
Another (slight) optimization is possible: if a certain node $i \in V$ transmits a *Test*-message along some edge $e$ and it receives a message *Connect*($L$) with $L < ln_i$ along this edge before it has actually received a response to that *Test*-message, then there is no need to wait for this response. In this case, $i$ would always receive a *Reject*-message afterwards. Consequently, node $i$ can, in this case, continue its search for the minimum-weight outgoing adjacent edge without waiting for a response to the *Test*-message. The node $j$ at the other end of $e$ could then as well ignore the *Test*-message in such a situation, i.e., if it attempts to process a message *Test*($l$, $f$) with $l < ln_i$ received along an edge in the state *branch*.

# 7 Conclusion

Correctness of the distributed minimum-weight spanning tree algorithm of Gallager, Humblet, and Spira [GHS83] has been proved. The strategy adopted in this paper in order to prove that the spanning tree algorithm meets its specification is to start with some sequential program which constructs the minimum-weight spanning tree, to refine, as described in [B88] and [CM88], parts of this program until distributed programs are obtained, and finally to combine these programs in order to obtain a distributed counterpart of the initial sequential program. The latter combinations have been accomplished by repeatedly applying the principle for sequentially phased reasoning about concurrently performed (sub)tasks, cf. [SR89a, SR89b]. These applications have shown that one can obtain from programs solving certain subtasks another program which solves the whole task, as if the subtasks are performed sequentially, even when these subtasks are performed repeatedly and concurrently by expanding groups of nodes. In addition, it has been shown that our principle can cope with with the phenomenon that tasks performed by one group of nodes are disturbed temporarily by interference of another group of nodes. For this reason invariants play an important role for our principle, since they allow one to prove interference-freedom of specifications. A future paper will show that such invariants can be generated during the design phase of programs.

## References


[AFR80] Apt K.R., Francez N., and de Roever W.P., A proof system for communicating sequential processes, ACM TOPLAS, 2-3 (1980).

[B88] Back R.J.R., A calculus of refinements for program derivations, Acta Informatica 25 (1988).

[BK83] Back R.J.R. and Kurki-Suonio, Decentralization of process nets with centralized control, Proc. of the ACM Symp. on Principles of Distr. Comp. (1983).

[CG88] Chou C.T. and Gafni E., Understanding and verifying distributed algorithms using stratified decomposition, Proc. of the ACM Symp. on Principles of Distr. Comp. (1988).

[CM88] Chandy K.M. and Misra J., Parallel program design: a foundation, Addison-Wesley Publishing Company, Inc. (1988).

[D59] Dijkstra E.W., Two problems in connections with graphs, Numer. Math. 1, (1959).

[D76] Dijkstra E.W., A discipline of programming, Prentice-Hall (1975).

[DS80] Dijkstra E.W. and Scholten C.S., Termination detecting for diffusing computations, Information Processing Letters 1-4 (1980).

[E79] Even S., Graph algorithms, Computer Science Press, Inc.(USA), (1979).

[F67] Floyd R.W.., Assigning meaning to programs, Mathematical aspects of computer science, AMS (1967).

[Fr80] Francez N., Distributed termination, ACM-TOPLAS, 2-1 (1980).

[G81] Gries D., The science of programming, Springer Verlag (1981).

[GHS83] Gallager R.T., Humblet P.A., and Spira P.M., A distributed algorithm for minimum-weight spanning trees, ACM TOPLAS, 5-1 (1983).

[H69] Hoare C.A.R., An axiomatic basis for computer programming, CACM, 12, (1969).

[H78] Hoare C.A.R., Communicating sequential processes, Comm. ACM, 21-8, (1978).

[Hu83] Humblet P.A., A distributed algorithm for minimum-weight directed spanning trees, IEEE Trans. on Comm., 31-6 (1983).

[K56] Kruskal J.B., On the shortest spanning subtree of a graph an the traveling salesman problem, Proc. Am. Math. Soc., 7, (1956).

[L83] Lamport L., Specifying concurrent modules, ACM TOPLAS, 5-2 (1983).

[MC81] Misra J. and Chandy K.M., Proofs of network of processes, IEEE. Trans. on Softw. Eng. 7 (1981).

[MP83] Manna Z. and Pnueli A., Verification of concurrent programs: A temporal proof system, Foundations of computer science IV, part 2, MC-tracts 159 (1983).

[MS79] Merlin P.M. and Segall A., A failsafe distributed routing protocol, IEEE Trans. on Comm., 27-9 (1979).

[OG76] Owicki S.S. and Gries D., An axiomatic proof technique for parallel programs, Acta Informatica 6 (1976).

[P89] Partsch H.A., Specification and transformation of programs -A formal approach to software development-, Springer-Verlag (1989, to appear).

[Se82] Segall A., Decentralized maximum-flow algorithms, Networks 12 (1982).

[Se83] Segall A., Distributed network protocols, IEEE Trans. on Inf. Theory. IT29-1 (1983).

[SR87] Stomp F.A. and de Roever W.P., A correctness proof of a distributed minimum-weight spanning tree algorithm (extended abstract), Proc. of the 7th ICDCS (1987).

[SR88] Stomp F.A. and de Roever W.P., A formalization of sequentially phased intuition in network protocols, Internal Report 88-15, University of Nijmegen (1988).

[SR89a] Stomp F.A. and de Roever W.P., Designing distributed algorithms by means of formal sequentially phased reasoning (extended abstract), To appear in the Proc. of the third International Conference on Distributed algorithms (1989).

[SR89b] Stomp F.A. and de Roever W.P., Designing distributed algorithms by means of formal sequentially phased reasoning, Internal Report 89-8, University of Nijmegen (1989).

[SS84] Schlichting R.D. and Schneider F.B., Using message passing for distributed programming, Proof rules and disciplines, ACM TOPLAS 6-3 (1984).

[WLL88a] Welch J.L., Lamport L., and Lynch N.A., A lattice-structured proof of a minimum spanning tree algorithm (extended abstract), Proc. of the ACM Symp. on Principles of Distr. Comp. (1988).

[WLL88b] Welch J.L., Lamport L., and Lynch N.A., A lattice-structured proof of a minimum spanning tree algorithm (full paper), Technical Report MIT (1988).

[ZS80] Zerbib F.B.M. and Segall A., A distributed shortest path protocol, Internal Report EE-395, Technion-Israel Institute of Technology, Haifa, Israel (1980).

[ZRE85] Zwiers J., de Roever W.P., and Emde Boas P., Compositionality and concurrent networks: soundness and completeness of a proof system, LNCS 194 (1985).

[Z89] Zwiers J., Compositionality, concurrency and partial correctness: proof theories for network of processes, and their connection, LNCS (1989).

# CHAPTER 5



Reprinted, with permission, from *Information and Computation*, Vol. 82, no. 3, 1989, pp. 278-322.

# The $\mu$-Calculus as an Assertion-Language for Fairness Arguments

F. A. STOMP

*University of Nijmegen, Department of Computer Science, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands*

AND

W. P. DE ROEVER AND R. T. GERTH

*Eindhoven University of Technology, Department of Computer Science and Mathematics, POB 513, 5600 MB Eindhoven, The Netherlands*

Various principles of proof have been proposed to reason about fairness. This paper addresses—for the first time—the question in what formalism such fairness arguments can be couched. To wit: we prove that Park's monotone first-order $\mu$-calculus, augmented with constants for all recursive ordinals can serve as an assertion-language for proving fair termination of do-loops. In particular, the weakest precondition for fair termination of a loop w.r.t. some postcondition is definable in it. The relevance of this result to proving eventualities in the temporal logic formalism of Manna and Pnuelis (*in* "Foundations of Computer Science IV, Part 2," Math. Centre Tracts, Vol. 159, Math. Centrum, Amsterdam, 1983) is discussed. © 1989 Academic Press, Inc.

## 1. MOTIVATION

Fairness is the defining property of good schedulers. The very notion of fairness presumes some kind of (metaphorical) competition for some shared resource(s). This competition is settled by arbitration, resulting in synchronization of competitor and resource. One speaks of a fair scheduling mechanism when this arbitration meets certain standards. Roughly, a scheduling discipline for a set of processes is called fair, whenever, inside a process, one or more (constituent) agents are "sufficiently often" allowed to compete for some shared resource, one of these agents is eventually scheduled for synchronization with that resource. Different notions of fairness can be distinguished according to their specification of what "sufficiently often" means, of their identification of resources, and of sets of agents inside processes, and of when these agents are considered to compete.

0890-5401/89 $3.00

The present paper concentrates on that notion of fairness, which prescribes that "an action which is infinitely often enabled is eventually taken." Here, sufficiently often is interpreted as infinitely often; the set of agents are singleton sets; the actions are guarded statements of guarded commands; an action is *enabled* (allowed to compete) whenever its guard evaluates to true; and whenever in a guarded selection all guards evaluate to false this selection is considered to be waiting, i.e., repeated execution results in (re-)evaluation of its guards (and possibly, in execution of a command guarded by a true guard), and not in failure upon its first execution as in sequential programming (Manna and Pnueli, 1983).

This notion of fairness is linked with the interleaving model of concurrency to remedy the following deficiency. Since the only requirement in the interleaving model is a syntactic one, namely, that actions from every process continue to be nondeterministically interleaved (sequentialized) as long as that process has not terminated, this requirement is also fulfilled for an interleaving which systematically selects re-evaluation of the guards of a waiting guarded selection when these happen to be false and which never selects execution of that selection when these guards have become true (due to some interleaved action of another process).

That is, in the interleaving model for concurrency, guards may be systematically selected for evaluation at the wrong moments. Now this behaviour does not occur in case every process has its own active processor (which notices when guards evaluate to true). Thus, the nondeterministically interleaved sequential execution of processes need not necessarily lead to the same result as the concurrent execution of those processes on separate processors. Yet we want to maintain the interleaving model of concurrency as our model for the concurrent execution of processes since this is the only model upon which successful verification theories have been built (other models for reasoning about correctness properties of concurrent processes are always obtained from this model by introducing equivalence relations and congruences). In this we succeed by imposing as an extra requirement the fairness requirement above.

Next, nearing the focus of this paper, the interaction between fairness and the interleaving model must be examined.

### *How Does One Deduce Properties in the Resulting Model?*

The properties of interest always contain eventualities which are enforced by the assumption of fairness. Pure invariances, i.e., properties which are invariant during execution, are not influenced by postulating fairness as an extra requirement and can be derived using more traditional methods.

The state of art offers the following picture: Let $\psi$ denote some state formula, i.e., $\psi$ is a direct property of program states not requiring temporal operators such as $\Diamond$ for its expression. To establish that for a concurrent program $\psi$ eventually holds, the following strategy is taken:

(1) Amongst the concurrent processes a distinction is made between those processes—in Manna and Pnueli's (1983) terminology dubbed *helpful* processes—whose execution brings satisfaction of $\psi$ always nearer, and those processes that do not do so, i.e., whose execution possibly does not bring satisfaction of $\psi$ any nearer, called *steady* (or unhelpful) processes.

(2) It must be proved that systematically avoiding execution of any helpful process either leads to an interleaving of steady processes which does not satisfy fairness, i.e., is unfair, since infinitely often a helpful process is enabled but not taken, or, due to some nondeterministic choice of a steady process in the interleaving, does bring satisfaction of $\psi$ eventually nearer or even establishes $\psi$.

Essential here is that upon closer inspection part (2) above requires application of the same strategy to a syntactically simpler program: just remove the helpful processes from the original program and prove that eventually one of the following holds: $\psi$, getting nearer to $\psi$ or, a helpful process is enabled.

As a preparation for a technical formulation of this strategy, we first introduce a number of auxiliary notions (Manna and Pnueli, 1983). Let $P \equiv P_1 \parallel \cdots \parallel P_n$ be some program with $n \geqslant 1$.

Assume that both $\phi$ and $\phi'$ are state formulae.

—For $i$ satisfying $1 \leqslant i \leqslant n$, we say that $P_i$ leads from $\phi$ to $\phi'$ when every state transition in $P_i$ establishes $\phi'$ provided $\phi$ is satisfied first.

—We say that $P$ leads from $\phi$ to $\phi'$ when for all $i$, $1 \leqslant i \leqslant n$, $P_i$ leads from $\phi$ to $\phi'$.

A technical formulation of the above-mentioned strategy requires the introduction of well-founded sets and looks as follows (Manna and Pnueli, 1983):

THE WELL-FOUNDED LIVENESS PRINCIPLE WELL. Let $\mathfrak{M} = (A, \leqslant)$ be a well-founded ordered structure. Let $\phi(\alpha)$ be a parametrized state formula over $A$, where $\alpha$ intuitively expresses how far establishing $\psi$ is. Let $h: A \to \{1, ..., n\}$ be a helpfulness function identifying for each $\alpha \in A$ the helpful process $P_{h(\alpha)}$ for states satisfying $\phi(\alpha)$.

(A) $\vdash P$ leads from $\phi(\alpha)$ to $[\psi \vee (\exists \beta \leqslant \alpha \cdot \phi(\beta))]$

(B) $\vdash P_{h(\alpha)}$ leads from $\phi(\alpha)$ to $[\psi \vee (\exists \beta < \alpha \cdot \phi(\beta))]$

(C) $\vdash \phi(\alpha) \supset \Diamond [\psi \vee (\exists \beta < \alpha \cdot \phi(\beta)) \vee \text{Enabled}(P_{h(\alpha)})]$

---

$\vdash (\exists \alpha \cdot \phi(\alpha)) \supset \Diamond \psi.$

The soundness proof of this rule requires induction over well-founded sets.

Conversely, given the fact that $\Diamond\psi$ is valid, (naive) set theory is used to argue the existence of the required auxiliary quantities, i.e., the well-founded ordered structure $\mathfrak{M}$, the *ranking predicate* $\phi(\alpha)$, and the helpfulness function $h$, which satisfy clauses (A), (B), (C), so that for each such $\psi$, WELL can always be applied. This proves that WELL is *semantically complete.*

Manna and Pnueli (1983) even prove that, for certain classes of formulae, their temporal logic formalism is complete relative to the set of temporal formulae valid in the given domain interpretation. Typically, their proof shows that the reasoning about temporal assertions concerning the execution sequences of programs can be reduced to the reasoning about assertions concerning the states of programs, the so-called state properties.

Now we are ready to ask the one question this paper is about: How do these results help us if we are sure that $\Diamond\psi$ holds and want to apply the rule above to verify $\Diamond\psi$? The answer is: not much.

Questions such as:

—How does one obtain the appropriate well-founded ordered structure $\mathfrak{M}$?

—How does one express, and reason about, the helpfulness function $h$ and the ranking predicate $\phi(\alpha)$?

—In general, which assertion-language should be used to establish hypotheses (A), (B), (C) of WELL?

are not answered by the above results, since the reasoning about state properties is not formalized in Manna and Pnueli (1983).

The present paper suggests a direction to answer these questions, by concentrating on these problems as they occur when proving termination of do-loops under the above fairness assumptions, i.e., fair termination of do-loops. That this does not lead to oversimplification follows from the fact that the same auxiliary quantities, with comparable objectives, occur in the rule whose expression and use we shall investigate (Grümberg, Francez, Makowsky, and de Roever, 1981).

THE WELL-FOUNDED LIVENESS PRINCIPLE FOR LOOPS—ORNA'S RULE. Let $\mathfrak{M} = (W, \leqslant)$ be a well-founded structure. Let $\pi: W \to$ (States $\to \{true, false\}$) be a predicate, and $q$ be a state predicate. Let for $w \in W$, with $w$ not minimal (denoted by $0 < w$), be given pairwise disjoint sets $D_w$ and $St_w$, such that $D_w \neq \emptyset$ and $D_w \cup St_w = \{1, ..., n\}$:

(a) $\vdash [\pi(w) \wedge w > 0 \wedge b_j]\ S_j\,[\exists v < w \cdot \pi(v)]$, for all $j \in D_w$

(b) $\vdash [\pi(w) \wedge w > 0 \wedge b_j]\ S_j\,[\exists v \leqslant w \cdot \pi(v)]$, for all $j \in St_w$

$$\begin{array}{ll} \text{(c)} & \vdash [\pi(w) \wedge w > 0]^* \, [\square_{i \in St_w} b_i \wedge \bigwedge_{j \in D_w} \neg b_j \rightarrow S_i] \, [\mathit{true}] \\ \text{(d)} & \vdash r \supset (\exists v, \pi(c)) \\ & \vdash (\pi(w) \wedge w > 0) \supset \bigvee_{i=1}^{n} b_i \\ & \vdash \pi(0) \supset ((\bigwedge_{i=1}^{n} \neg b_i) \wedge q) \\ \hline & \vdash [r]^* \, [\square_{i=1}^{n} b_i \rightarrow S_i][q]. \end{array}$$

Note, when comparing Orna's rule with WELL, that the commands $S_i$ act as state transitions. Since in Orna's rule the assignment $w \rightarrow (D_w, St_w)$ for $w > 0$ merely generalizes WELL's notion of helpfulness function, the same kind of auxiliary quantities are required to apply both rules.

This paper proves that to express and reason about $\mathfrak{M}$, $\pi$, and the assignment $w \rightarrow (D_w, St_w)$ for $w > 0$ and $w \in W$, a slight extension is required of the formalism used to prove termination of recursive procedures, Park's $\mu$-calculus (Hitchcock and Park, 1973; Park, 1969).

Finally we note that, historically, two rules have been formulated to prove fair termination of nondeterministic programs: Orna's rule (Grümberg *et al.*, 1981) and the LPS-rule (Lehmann *et al.*, 1981). Both these rules model, each in their own way, a specific intuition related to the notion of eventuality implied by fairness assumptions. For fairly terminating loops they have been proved to be equivalent (Grümberg *et al.*, 1981), but the LPS-rule also applies to proving fair termination of concurrent processes.

This article is organized as follows: Section 1 contains the motivation for this paper; Section 2 specifies the programming language used in this paper. In this programming language, we restrict ourselves to sequences of assignments and to commands in which nested repetitions are not allowed. Section 3 discusses various semantics for this programming language. In Sections 4 and 5 the proof system and the assertion-language, i.e., the monotone $\mu$-calculus, are dealt with. A term in the assertion-language, which expresses fair termination of a repetition is constructed in Section 6. Completeness and soundness of the proof system are proved in Sections 7 and 8. In Section 9 we drop the restriction that we imposed w.r.t. the nesting of repetitions and outline how to deal with the more general case in which nested repetitions are allowed as commands. Finally Section 10 contains the conclusion.

## 2. The Language of Guarded Commands

In this section we describe the syntax of the programming language used throughout this paper. In the next section various semantics for this language are defined.

The syntax is specified below using the standard BNF-notation (braces enclose a repeated item, that may occur zero or more times). We do not specify the structure of variables and (boolean) expressions. Expressions are assumed to be terms in an underlying signature containing constant, function, and predicate symbols. We shall only use simple variables in the remainder of this paper.

DEFINITION 2.1 (Syntax of the programming language). Start with some signature. The language of guarded commands, LGC, is defined by:

$\langle$command$\rangle$ ::= $\langle$repetition$\rangle$ | $\langle$simple command$\rangle$.
$\langle$simple command$\rangle$ ::= $\langle$assignment$\rangle$ |
$\langle$simple command$\rangle$; $\langle$simple command$\rangle$.
$\langle$assignment$\rangle$ ::= $\langle$variable$\rangle$ := $\langle$expression$\rangle$.
$\langle$repetition$\rangle$ ::= *[ { $\square$ $\langle$selection$\rangle$ } ].
$\langle$selection$\rangle$ ::= $\langle$guard$\rangle$ $\rightarrow$ $\langle$simple command$\rangle$.
$\langle$guard$\rangle$ ::= "a quantifier-free (boolean) expression."

We identify *[ ] with the assignment $x := x$ (**skip**). In the remainder of this paper, we shall often abbreviate $*[\square b_1 \rightarrow S_1 \square \cdots \square b_n \rightarrow S_n]$ to $*[\square_{i=1}^n b_i \rightarrow S_i]$.

The main differences between the language as described above and that of Dijkstra's are that, in our language, guarded selections are not allowed as commands and that in a repetition $*[\square_{i=1}^n b_i \rightarrow S_i]$, the $S_i$ never contain repetitions ($i = 1, ..., n$). In Section 9, it is shown how to deal with fairness issues when the latter restriction is dropped.

In the sequel we also need the notion of a direction of a repetition $*[\square_{i=1}^n b_i \rightarrow S_i]$ with $n \geqslant 1$.

DEFINITION 2.2 (Directions of repetition). Let $S \equiv *[\square_{i=1}^n b_i \rightarrow S_i]$ be a repetition with $n \geqslant 1$. For $i = 1, ..., n$, $b_i$; $S_i$ is called the $i$th direction of $S$.

## 3. SEMANTICS

In this section we define four semantics for the language of Section 2. Two of them are defined without consideration of fairness constraints. The other ones are defined when such fairness constraints are imposed. The first semantics fitting for partial correctness is defined using relations, since non-determinism is involved. To reason about (nondeterministic) termination, we introduce the notions of an execution sequence of a repetition and of nondeterministic divergence of a repetition. Then the partial correctness semantics is extended to fit for total correctness.

Thereafter, we discuss two important fairness constraints, viz., strong fairness and unconditional fairness. These constraints lead to the notions of a strongly fair or unconditionally fair execution sequence of a repetition, of strongly fair or unconditionally fair divergence of a repetition from some state $\xi$, and of strongly fair or unconditionally fair termination of repetition.

The relation between nondeterministic termination, strongly fair termination, and unconditionally fair termination of a repetition is discussed. The third semantics in this section is defined taking strong fairness into account; the fourth one takes unconditional fairness into account.

### 3.1. *Preliminaries*

Before defining the various semantics for the language of Section 2, we first recapitulate a number of basic notions.

DEFINITION 3.1.1 (First-order structure). A first-order structure $\mathfrak{M}$ consists of

(a) a non-empty set, also referred to as a domain, denoted by $|\mathfrak{M}|$,

(b) a set of $n$-ary function symbols and a set of $n$-ary predicate symbols ($n \geqslant 0$), such that for each $n$-ary function symbol (resp. predicate symbol) there corresponds a $n$-ary function (resp. predicate) over $|\mathfrak{M}|$, and

(c) a set of constant symbols, corresponding to elements of $|\mathfrak{M}|$.

We assume the equality symbol "$=$" to be present as a binary predicate symbol, corresponding to the standard equality over $\mathfrak{M}$.

In the remainder of this section we assume that $\mathfrak{M}$ is some first-order structure, which contains all symbols that may appear in a program $S \in \mathrm{LGC}$. We adopt the convention to denote LGC by LGC($\mathfrak{M}$) in such a case.

DEFINITION 3.1.2 (State, enabledness, disabledness, state variant).

(a) A state is a function from the collection of all program variables to the domain of interpretation. $\xi$, $\xi_i$, $\xi'$, etc. are used to denote states. The set of all states is denoted by States. The value of the expression $e$ in state $\xi$ is denoted by $\xi(e)$. (We assume that the $\xi(e)$ is always defined!)

(b) If a guard $b$ evaluates to true in state $\xi$, i.e., $\xi(b)$ holds, we say that $b$ is enabled in state $\xi$; otherwise, $b$ is disabled in $\xi$.

(c) For a state $\xi$, a variable $x$, and an expression $e$, the state variant $\xi\{e/x\}$ is defined as usual: $\xi\{e/x\}(x) = \xi(e)$, and $\xi\{e/x\}(y) = \xi(y)$ if $x \not\equiv y$.

Next, we introduce the operator "$\circ$" denoting composition of relations.

DEFINITION 3.1.3 (Composition of relations). Let $A_1$, $A_2$, and $A_3$ denote sets. Assume that $R_1 \subseteq A_1 \times A_2$ and $R_2 \subseteq A_2 \times A_3$ are binary relations. Then $R_1 \circ R_2 \subseteq A_1 \times A_3$ is a binary relation, too. This relation satisfies: for all $a_1 \in A_1$, $a_3 \in A_3$, $(R_1 \circ R_2)(a_1, a_3)$ holds iff there exists some $a_2 \in A_2$ with $R_1(a_1, a_2)$ and $R_2(a_2, a_3)$.

### 3.2. *Partial Correctness*

We now associate with each program $S$ the (relational) semantics $R_S^{\text{part}} \subseteq \text{States} \times \text{States}$. Note that, due to nondeterminism, for input state $\xi$ and program $S$, there may be more than one output state or even infinitely many ones. If $S$ nowhere terminates when started in $\xi$ (in the semantics under discussion) there will be no output state, i.e., the set of output states is empty.

DEFINITION 3.2.1 (Partial correctness semantics).

(a) $S \equiv x := e : R_S^{\text{part}} = \{(\xi, \xi\{e/x\}) \mid \xi \text{ a state}\}$.

(b) $S \equiv S_1 ; S_2$, for simple commands $S_1$ and $S_2$: $R_S^{\text{part}} = R_{S_1}^{\text{part}} \circ R_{S_2}^{\text{part}}$.

(c) $S \equiv {}^*[\Box_{i=1}^n b_i \to S_i]$, for $n \geqslant 1$ and simple command $S_i$, $i = 1, \ldots, n$: Let $R_B = \{(\xi, \xi) \mid \xi \text{ a state satisfying } B\}$ for boolean expressions $B$ and let $b$ denote the formula $\bigvee_{i=1}^n b_i$. Define $R_{\bar{S}} = \bigcup_{i=1}^n (R_{b_i} \circ R_{S_i}^{\text{part}})$. Then $R_S^{\text{part}} = (\bigcup_{i=0}^{\infty} R_{\bar{S}}^i) \circ R_{\neg b}$, where $R_{\bar{S}}^i$ denotes the $i$-fold composition of the relation $R_{\bar{S}}$ with itself.

Observe that for repetitions $S \equiv {}^*[\Box_{i=1}^n b_i \to S_i]$, $R_S^{\text{part}}$ contains the pairs $(\xi, \xi)$ for $\xi$ satisfying $\xi \models \bigwedge_{i=1}^n \neg b_i$. This means that $S$ "immediately" terminates if $S$ is executed in an initial state in which none of the guards is enabled.

DEFINITION 3.2.2 ($[p]\, S[q]_{\text{part}}$). Let $p$ and $q$ denote assertions in an assertion-language containing all program variables, terms, and boolean expressions over $\mathfrak{M}$. Let $S \in \text{LGC}(\mathfrak{M})$. Then we define $\mathfrak{M} \models [p]\, S[q]_{\text{part}}$ iff $\mathfrak{M} \models \forall \xi, \xi'[(p(\xi) \wedge R_S^{\text{part}}(\xi, \xi')) \supset q(\xi')]$ (partial correctness). I.e., $\mathfrak{M} \models [p]\, S[q]_{\text{part}}$ holds iff "for all input states $\xi$ satisfying $p$ the following holds: if $S$ terminates when started in $\xi$, then the output state satisfies $q$."

### 3.3. *Total Correctness*

Next, to reason about termination, we add to the set of states a special state $\perp$, standing for divergence. As usual, the state variant $\perp\{e/x\}$ is defined to be $\perp$. For an assertion $p$, $p(\perp)$ is defined to be false, i.e., $p$ never holds in $\perp$. In the sequel we assume $\perp$ to be present in States.

DEFINITION 3.3.1 (Total correctness semantics, execution sequences of repetitions, nondeterministic divergence of a repetition from a state). Define the relation $R'_S$, for $S \in \mathrm{LGC}(\mathfrak{M})$ as follows:

(a) $R'_S = R_S^{\mathrm{part}} \cup \{(\perp, \perp)\}$, if $S \equiv x := e$.

(b) $R'_S = (R'_{S_1} \circ R'_{S_2})$, if $S \equiv S_1; S_2$ and both $S_1$ and $S_2$ are simple. To define $R'_S$ for repetitions $S$, the notion of an execution sequence of $S$ is introduced:

(c) an execution sequence of a repetition $S \equiv *[\Box_{i=1}^{n} b_i \to S_i]$, $n \geqslant 1$, is a maximal sequence of states $\xi_0 \to^{i_0} \xi_1 \to^{i_1} \xi_2, \ldots$ such that $(R_{b_k} \circ R'_{S_k})(\xi_j, \xi_{j+1})$ holds for all $j, k$ satisfying $j \geqslant 0$ and $k = i_j$ with $1 \leqslant k \leqslant n$. The sequence is considered to be maximal if it cannot be extended, i.e., it is either infinite or ends with some state $\xi_k$ satisfying $\bigwedge_{i=1}^{n} \neg b_i$.

(d) We say that a repetition $S$ can diverge nondeterministically from $\xi$ if there exists an infinite execution sequence of $S$ starting in $\xi$.

(e) For $S \equiv *[\Box_{i=1}^{n} b_i \to S_i]$ with $n \geqslant 1$ and simple commands $S_i$ $(i = 1, \ldots, n)$, define $R'_S = R_S^{\mathrm{part}} \cup \{(\xi, \perp) \mid S$ can diverge nondeterministically from $\xi\} \cup \{(\perp, \perp)\}$.

DEFINITION 3.3.2 (Nondeterministic termination, $[p]\,S[q]_t$). For $S \in \mathrm{LGC}(\mathfrak{M})$ and assertions $p, q$ as above:

(a) Termination of a (nondeterministic) program $S$ is straightforwardly defined as $\forall \xi \neq \perp \cdot \neg R'_S(\xi, \perp)$.

(b) $\mathfrak{M} \models [p]\,S[q]_t$ iff $\mathfrak{M} \models \forall \xi, \xi'[(p(\xi) \wedge R'_S(\xi, \xi')) \supset q(\xi')]$ (total correctness). I.e., $\mathfrak{M} \models [p]\,S[q]_t$ holds iff "$S$ always terminates in a state satisfying $q$, provided execution of $S$ started in a state satisfying $p$."

### 3.4. *Strong Fairness and Unconditional Fairness*

Termination of a program $S$ has been defined as $\forall \xi \neq \perp.\ \neg R'_S(\xi, \perp)$. This is, however, a rather strong requirement. Consider, e.g., Dijkstra's (1976) random number generator: $S_0 \equiv *[b \to x := x + 1 \ \Box\ b \to b := \mathit{false}]$. $S_0$ need not necessarily terminate if started in a state $\xi$ such that $\xi(b)$ holds, because its execution may be governed by an extremely one-sided scheduler that consistently refuses to execute the second direction of $S_0$, i.e., $b; b := \mathit{false}$, in any iteration.

Consequently, various constraints on schedulers have been proposed which prohibit schedulers to neglect the execution of directions under certain circumstances. Termination of a repetition is considered relative to a set of schedulers thus constrained.

Before presenting two important constraints or fairness assumptions on such schedulers, viz., *strong fairness* and *unconditional fairness* (Apt *et al.*, 1984; Lehman *et al.*, 1981), we first introduce the notions of enabledness and disabledness of directions of a repetition.

DEFINITION 3.4.1 (Enabledness and disabledness of directions). Let $S \equiv *[\square_{i=1}^{n} b_i \rightarrow S_i]$ be a repetition. Assume that $\xi_0 \rightarrow^{i_0} \xi_1 \rightarrow^{i_1} \cdots$ is an execution sequence of $S$. For state $\xi_m$, $m \geqslant 0$, occurring in this sequence we say that the $i$th direction of $S$ is enabled in $\xi_m$ if $\xi_m(b_i)$ holds, where $1 \leqslant i \leqslant n$; otherwise the $i$th direction of $S$ is disabled in $\xi_m$.

DEFINITION 3.4.2 (Strongly fair execution sequences, strongly fair termination, strongly fair divergence of repetitions).

(a) An execution sequence of a repetition $S$ is strongly fair, either if it is finite or if it is infinite and every direction of $S$ which is infinitely often enabled in this sequence is chosen infinitely often along the sequence.

(b) A repetition terminates strongly fair if it admits no infinite strongly fair execution sequences.

(c) A repetition diverges strongly fair from state $\xi$ if it admits an infinite strongly fair execution sequence starting in $\xi$.

Observe that, while the above program, $S_0$, admits infinite computations, none of them is strongly fair; i.e., $S_0$ terminates strongly fair.

In the sequel, we also need the notion of unconditional fairness, that does not take enabledness and disabledness of directions into account.

DEFINITION 3.4.3 (Unconditionally fair execution sequences, unconditionally fair termination, unconditionally fair divergence of a repetition).

(a) An execution sequence of a repetition is unconditionally fair, either if it is finite or if it is infinite and every direction is chosen infinitely often along the sequence.

(b) A repetition terminates unconditionally fair if it admits no infinite unconditionally fair execution sequences.

(c) A repetition diverges unconditionally fair from state $\xi$ if it admits an infinite unconditionally fair execution sequence starting in $\xi$.

The program $S_1 \equiv *[x=0 \rightarrow x := 1 \ \square\ x=1 \rightarrow x := x]$ does admit infinite strongly fair computations, but no unconditionally fair ones.

Other examples of unconditionally fair and strongly fair terminating programs can be found in Grümberg *et al.* (1983). We should remark here that some authors use a different terminology. In Lehmann *et al.* (1981) the names impartiality (resp. fair) are used instead of unconditionally fair (resp. strongly fair).

The relation between nondeterministic termination, strongly fair termination, and unconditionally fair termination of a repetition is given in the following:

THEOREM 3.4.4 (Relation between unconditionally fair, strongly fair, and nondeterministic termination). *For each repetition* $S$,

(i) *$S$ terminates nondeterministically $\Rightarrow$ $S$ terminates strongly fair.*

(ii) *$S$ terminates strongly fair $\Rightarrow$ $S$ terminates unconditionally fair.*

*Proof.* (i) and (ii) immediately follow from the definitions above. Observe that the examples above show that the implications are proper.

We now proceed to define other semantics, taking fairness assumptions into account. The meaning of a command $S$ under the assumption of strong fairness is given by the relation $R_S^{\mathrm{sf}}$; under the assumption of unconditional fairness it is given by the relation $R_S^{\mathrm{uf}}$.

DEFINITION 3.4.5 (Semantics under fairness assumptions). For simple commands $S$, we simply define:

$R_S^{\mathrm{uf}} = R_S^{\mathrm{sf}} = R'_S$,

and for repetitions $S \equiv {}^*[\square_{i=1}^n b_i \rightarrow S_i]$ with $n \geqslant 1$ and simple $S_i$, $i = 1, ..., n$:

$R_S^{\mathrm{uf}} = R_S^{\mathrm{part}} \cup \{(\xi, \bot) \mid S$ can diverge unconditionally fair from $\xi\} \cup \{(\bot, \bot)\}$ and

$R_S^{\mathrm{sf}} = R_S^{\mathrm{part}} \cup \{(\xi, \bot) \mid S$ can diverge strongly fair from $\xi\} \cup \{(\bot, \bot)\}$.

Next, termination of a program $S$ under fairness assumptions and validity of $[p]\, S[q]_s$ for $s \in \{\mathrm{uf}, \mathrm{sf}\}$ are defined.

DEFINITION 3.4.6 (Termination under fairness assumptions, $[p]\, S[q]_{\mathrm{sf}}$, and $[p]\, S[q]_{\mathrm{uf}}$). (a) A program $S$ terminates strongly fair, unconditionally fair, respectively, iff $\forall \xi \neq \bot \cdot \neg R_S^{\mathrm{sf}}(\xi, \bot)$, $\forall \xi \neq \bot \cdot \neg R_S^{\mathrm{uf}}(\xi, \bot)$, respectively, hold. (Cf. Definitions 3.4.2(b) and 3.4.3(b).)

(b) For $s \in \{\mathrm{uf}, \mathrm{sf}\}$, assertions $p$ and $q$, as above, and program $S$, we define

$$\mathfrak{M} \models [p]\, S[q]_s \quad \text{iff} \quad \mathfrak{M} \models \forall \xi, \xi' [(p(\xi) \wedge R_S^s(\xi, \xi')) \supset q(\xi')].$$

In the sequel $\xi$ denotes a state other than $\bot$, unless stated otherwise.

## 4. THE PROOF SYSTEM

We use a Hoare-like proof system. The axioms and rules are as follows:

(1) *assignment*

$$[p\{e/x\}]\, x := e\, [p];$$

(2) *composition*

$$\frac{[p]\, S_1[q],\ [q]\, S_2[r]}{[p]\, S_1; S_2[r]}, \qquad \text{for simple commands } S_1, S_2;$$

(3) *consequence*

$$\frac{p \supset p_1,\ [p_1]\, S[q_1],\ q_1 \supset q}{[p]\, S[q]};$$

(4) *Orna's rule* (see Section 1), for simple commands $S_i$ ($i = 1, ..., n$).

Note that we only consider repetitions under the assumption of strong fairness. However, Orna's rule can also be applied to ordinary terminating do-loops. In this case, one simply takes the sets $St_w$, $w \in W$ to be empty. We then obtain Harel's (1979) rule for terminating loops.

## 5. The Assertion-Language $L$

Our assertion-language is based on the $\mu$-calculus of Hitchcock and Park (1973; also Park, 1969), which is appropriate both to prove termination of recursive parameterless procedures (see de Bakker, 1980; Hitchcock and Park, 1973) and to express the auxiliary quantities associated with those proofs.

In this section, we first recapitulate the basic ideas on which the $\mu$-calculus is based and introduce some fixed point definitions that are needed in Sections 6 and 7. In particular, we express the domain of well-foundedness of a binary relation as a $\mu$-term. The term expressing the non-existence of infinite strongly fair execution sequences of a loop, see Section 6, will be a more complicated variant of that $\mu$-term.

After introducing the assertion-language $L$ used throughout the remainder of this paper, we define validity of formulae in $L$. As is usual in completeness proofs, we shall need the ability to encode finite sequences. In this, we base ourselves on Moschovakis (1974).

As is argued in Apt and Plotkin (1985), fairness arguments require the use of recursive ordinals. For this reason we introduce the notion of an ordinal acceptable structure (see Definition 5.5.3). Relative to such structures completeness will be shown in Section 7.

### 5.1. *Preliminaries*

The $\mu$-calculus is based on Knaster and Tarski's theorem (Tarski, 1955).

Theorem 5.1.1 (Knaster–Tarski theorem). *Let* $(A, \sqsubseteq)$ *be a complete lattice and* $F\colon A \to A$ *a monotonic function; in fact a cpo suffices. Then* $F$ *has a least fixed point, denoted by* $\mu a \cdot [F(a)]$, *meaning that*

(i) *$F(\mu a \cdot [F(a)]) = \mu a \cdot [F(a)]$, i.e., $\mu a \cdot [F(a)]$ is a fixed point of $F$.*

(ii) *if there exists some $b \in A$ such that $F(b) = b$, then $\mu a \cdot [F(a)] \sqsubseteq b$, i.e., $\mu a \cdot [F(a)]$ is the* least *fixed point of $F$.*

Using the notation as above, $\mu a \cdot [F(a)]$ is unique since the partial ordering $\sqsubseteq$ is anti-symmetric. In the sequel, we refer to property (i) formulated in Theorem 5.1.1 as the fixed point property.

LEMMA 5.1.2 (Characterizations of least fixed points). *There are several ways to regard least fixed points. Using the notation as above, first,*

(a) *$\mu a \cdot [F(a)] = \bigsqcap \{x \in A \mid F(x) = x\} = \bigsqcap \{x \in A \mid F(x) \sqsubseteq x\}$, where $\bigsqcap$ denotes the infimum. A proof of this can be found in de Bakker* (1980).

*Second, the least fixed point can be obtained by iterating $F$ into the transfinite ordinals.*

(b) *Define for each ordinal $\lambda$:*

$$F^0(x) = x,$$

$$F^\lambda(x) = F\left(\bigsqcup_{\beta < \lambda} F^\beta(x)\right), \qquad \textit{if} \quad \lambda \neq 0.$$

Here $\bigsqcup$ denotes the supremum. Let $\perp_A$ denote $A$'s least element, which exists since $A$ is a complete lattice. Then $\mu a \cdot [F(a)] = F^\alpha(\perp_A)$ for some ordinal $\alpha$. For a proof, we refer the reader to Moshovakis (1974). Clearly, if $\mu a \cdot [F(a)] = F^\alpha(\perp_A)$ holds, then for all $\beta \geqslant \alpha$, $\mu a \cdot [F(a)] = F^\beta(\perp_A)$ holds, too.

### 5.2. *Fixed Point Definitions*

Next, we introduce some fixed point definitions.

DEFINITION 5.2.1 ($R \rightarrow p$, $R \circ p$). Let $R$ be a binary relation over some set and let $p$ be a predicate on the same set. Define

(i) $R \rightarrow p$ by $(R \rightarrow p)(x)$ iff $\forall x' \cdot [R(x, x') \supset p(x')]$, and its dual

(ii) $R \circ p$ by $\neg(R \rightarrow \neg p)$. So $(R \circ p)(x)$ holds iff $\exists x' \cdot [R(x, x') \wedge p(x')]$.

Since the collection of predicates ordered by $p \sqsubseteq q$ iff $p \supset q$ forms a complete lattice with *false* as the least element, and $R \rightarrow p$, as well as $R \circ p$, is monotonic in $p$, $\mu p \cdot [R \rightarrow p]$ exists.

THEOREM 5.2.2 (Domain of well-foundedness of a binary relation $R$, $\mu p \cdot [R \rightarrow p]$). *Let $R$ be a binary relation over some set. Then $\mu p \cdot [R \rightarrow p]$ describes the domain of well-foundedness of $R$; i.e., for all $x$ the following is* *satisfied*: $\mu p \cdot [R \rightarrow p](x)$ *holds iff there exists no infinite sequence* $x_0, x_1, x_2, \ldots$ *with* $x = x_0$ *and* $R(x_i, x_{i+1})$ $(i \geqslant 0)$.

*Proof.* ($\Rightarrow$) Define $\tau(p) = R \rightarrow p$. Observe that $\mu p \cdot [R \rightarrow p] = \tau^{\alpha}(\textit{false})$ holds for some ordinal $\alpha$. Consequently, it suffices to show that for all $x$: if $\tau^{\alpha}(\textit{false})(x)$ holds, then there exists no infinite sequence $x_0, x_1, x_2, \ldots$ with $x = x_0$ and $R(x_i, x_{i+1})$ for $i \geqslant 0$.

Using induction on $\beta$, we prove that for all $\beta \leqslant \alpha$ the following holds: $\tau^{\beta}(\textit{false})(x) \Rightarrow$ there is no infinite sequence $x_0, x_1, x_2, \ldots$ with $x = x_0$ and $R(x_i, x_{i+1})$ $(i \geqslant 0)$ holds.

Induction basis. $\beta = 0$: trivial.

Induction hypothesis. Suppose that the implication holds for all $\lambda < \beta$.

Induction step. For $\beta \neq 0$, we have

$$\begin{aligned} \tau^{\beta}(\textit{false})(x) &\Leftrightarrow \Big(R \rightarrow \bigsqcup_{\lambda < \beta} \tau^{\lambda}(\textit{false})\Big)(x) \\ &\Leftrightarrow \forall x' \cdot \Big[R(x, x') \Rightarrow \Big(\bigsqcup_{\lambda < \beta} \tau^{\lambda}(\textit{false})\Big)(x')\Big]. \end{aligned}$$

So $\tau^{\beta}(\textit{false})(x)$ implies that for all $x'$ such that $R(x, x')$ no infinite "descending" sequence starting in $x'$ exists. This follows from the induction hypothesis. Then there is no infinite "descending" sequence starting in $x$.

($\Leftarrow$) To prove the other implication, assume that $\neg \mu p \cdot [R \rightarrow p](x)$ holds. By the fixed point property, $\neg(R \rightarrow \mu p \cdot [R \rightarrow p])(x)$ holds, too. So, there is an $x_1$ such that $R(x, x_1)$ and $\neg \mu p \cdot [R \rightarrow p](x_1)$. This process can be repeated ad infinitum, and we obtain an infinite "descending" sequence $x_0, x_1, x_2, \ldots$ such that $x = x_0$ and $R(x_i, x_{i+1})$ $(i \geqslant 0)$. ∎

If $F$ is a monotonic operator mapping predicates to predicates, then its greatest fixed point, $\nu p \cdot [F(p)]$, exists too. This is because the collection of predicates as defined above is a complete lattice. Moreover, the greatest fixed point is representable in terms of the $\mu$-operator. This follows from the following lemma whose proof can be found in de Bakker (1980).

LEMMA 5.2.3 (Representability of the greatest fixed point in $\mu$-terms).

$$\nu p \cdot [F(p)] \Leftrightarrow \neg \mu p \cdot \neg [F(p)\{\neg p/p\}].$$

Since $R \circ p$ is monotonic in $p$, $\nu p \cdot [R \circ p]$ exists. Using Lemma 5.2.3, we obtain the equivalences $\nu p \cdot [R \circ p] \Leftrightarrow \neg \mu p \cdot [\neg(R \circ \neg p)] \Leftrightarrow \neg \mu p \cdot [R \rightarrow p]$.

Recall that "$\circ$" denotes composition of relations. We adopt the convention that "$\circ$" has priority over "$\cup$." I.e., $R_1 \circ R_2 \cup R_3$ should be parsed as $(R_1 \circ R_2) \cup R_3$.

Let $R$ denote a binary relation over some set, and let $I$ denote the identity relation over the same set. It is easily seen that $F(X) = R \circ X \cup I$ is monotonic in $X$, where $X$ denotes a relation variable. So $F$'s least fixed point $\mu X \cdot [R \circ X \cup I]$ exists. In informal notation $\mu X \cdot [R \circ X \cup I] = I \cup R \cup R^2 \cup \cdots \cup R^n \cup \cdots$.

*Notation* 5.2.4 ($R^*$, $R^+$).

(a) We abbreviate $\mu X \cdot [R \circ X \cup I]$ to $R^*$, the relation obtained by composing $R$, zero or more times with itself.

(b) In the sequel, we shall also use $R^+$, the relation obtained by composing $R$ at least once with itself, as an abbreviation for $R \circ R^*$.

We then have

FACT 5.2.5. Let $R$ denote a binary relation over some set and $I$ the identity relation over the same set. The following holds:

(a) $I \subseteq R^*$, $R^+ \subseteq R^*$, $R^+ = R^* \circ R$.

(b) If $T$ denotes a binary relation and $T \subseteq R$, then $T^* \subseteq R^*$ and $R^* \circ T \subseteq R^*$.

## 5.3. *The Assertion Language L*

Let $\mathfrak{M}$ be some first-order structure. The first-order logic over $\mathfrak{M}$ is defined as usual. Now we extend this logic so as to be able to express fixed point definitions. For this an infinite set of $n$-ary predicate variables, $p, X, Y, \ldots$, is introduced for every $n \geqslant 0$. These predicate variables may appear in formulae, but may not be bound by quantifiers. These variables from the basis of the fixed point definitions. To ensure the existence of least (and greatest) fixed points, monotonicity has to be imposed. In fact, we introduce the notion of syntactic monotonicity of formulae, which implies their semantic monotonicity. In essence, this notion requires that each occurrence of the predicate variable $p$ that is to be bound by the least fixed point operator $\mu$ is within the scope of an even number of $\neg$-signs.

DEFINITION 5.3.1 (Syntactic monotonicity and syntactic anti-monotonicity). We inductively define sets $\mathrm{sm}(p)$ (resp. $\mathrm{sa}(p)$), denoting the class of formulae that are syntactically monotonic (resp. syntactically anti-monotonic) in a variable $p$:

(i) $\phi \in \mathrm{sm}(p)$, if $p$ does not occur free in $\phi$.

(ii) $\neg\phi \in \mathrm{sm}(p)$, if $\phi \in \mathrm{sa}(p)$.

(iii) $\phi_1 \supset \phi_2 \in \mathrm{sm}(p)$, if $\phi_1 \in \mathrm{sa}(p)$ and $\phi_2 \in \mathrm{sm}(p)$.

(iv) $\forall x\phi$, $\exists x\phi \in \mathrm{sm}(p)$, if $\phi \in \mathrm{sm}(p)$.

(v) $p \in \mathrm{sm}(p)$.

(vi) $\mu p_1 \cdot [\phi]$, $\nu p_1 \cdot [\phi] \in \mathrm{sm}(p)$, if $\phi \in \mathrm{sm}(p) \cap \mathrm{sm}(p_1)$.

(vii) (i)–(iv) with sm and sa interchanged.

(viii) $\mu p_1 \cdot [\phi]$, $\nu p_1 \cdot [\phi] \in \mathrm{sa}(p)$, if $\phi \in \mathrm{sa}(p) \cap \mathrm{sm}(p_1)$.

Under the usual ordering, $\phi_1 \sqsubseteq \phi_2$ iff $\phi_1 \supset \phi_2$, it can be proved by induction on the structure, i.e., the complexity of the formulae that syntactic monotonicity implies semantic monotonicity.

DEFINITION 5.3.2 (Assertion-language). The assertion-language $L(\mathfrak{M})$ over some structure $\mathfrak{M}$, is the smallest class $B$ such that

(i) $\phi$, $\mu p \cdot [\psi(p)]$, $\nu p \cdot [\psi(p)] \in B$, where $\phi$ and $\psi$ are first-order formulae over $\mathfrak{M}$, $\phi$ does not contain any free predicate variables and $\psi \in \mathrm{sm}(p)$.

(ii) if $\phi, \psi \in B$ then $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \supset \psi$, and $\neg\phi \in B$, too.

*Remark.* If in a formula $\mu p \cdot [\psi(p)]$ or $\nu p \cdot [\psi(p)]$, $p$ does not occur free in $\psi$, then we will often write $\psi$ instead. Note that formulae of the form $\mu p \cdot [\psi(p)]$, where $\psi$ contains a $\mu$-operator, are not allowed. However, we shall use such formulae, in which such a nesting of $\mu$-operators occurs, since they are representable in $L(\mathfrak{M})$, see Moschovakis (1974).

In the sequel we shall often abbreviate $L(\mathfrak{M})$ to $L$, when the structure $\mathfrak{M}$ is clear from the context.

## 5.4. *Validity of L-Formulae*

We next define validity of $L$-formulae. This definition is clear, except for the cases $\mu p \cdot [\psi(p)]$ and $\nu p \cdot [\psi(p)]$. Recall that $\mu p \cdot [\psi(p)]$ can be obtained by iteration. We now formalize this idea in the following construct by defining predicates $I_\psi^\beta$ for $\beta \geqslant 0$ "by iterating $\psi$ $\beta$ times from below."

DEFINITION 5.4.1 ($I_\psi^\beta$). For first-order formulae $\psi$ over $\mathfrak{M}$, $\psi \in \mathrm{sm}(p)$, we define $I_\psi^\beta$ for ordinals $\beta$ by

$$I_\psi^0 = \lambda \bar{x} \cdot \mathit{false},$$
$$I_\psi^\beta = \lambda \bar{x} \cdot \psi(\bar{x}, \textstyle\bigsqcup_{\alpha < \beta} I_\psi^\alpha) \text{ for } \beta \neq 0,$$
$$I_\psi = \lambda \bar{x} \cdot \textstyle\bigsqcup_{\alpha \geqslant 0} I_\psi^\alpha(\bar{x}).$$

By the monotonicity of $\psi$ the following holds (Moschovakis, 1974):

LEMMA 5.4.2 (Properties of $I_\psi^\beta$).

(i) $(\alpha \leqslant \beta) \Rightarrow (I_\psi^\alpha(\bar{x}) \Rightarrow I_\psi^\beta(\bar{x}))$;

(ii) *for some ordinal* $\kappa$: $I_\psi = I_\psi^\kappa = \bigsqcup_{\alpha < \kappa} I_\psi^\alpha$;

(iii) *$I_\psi$ is the least predicate $C$ satisfying $C(\bar{x}) \Leftrightarrow \psi(\bar{x}, C)$; i.e., $I_\psi(\bar{x}) \Leftrightarrow \psi(\bar{x}, I_\psi)$ and if $C$ satisfies $C(\bar{x}) \Leftrightarrow \psi(\bar{x}, C)$, then $I_\psi(\bar{x}) \Rightarrow C(\bar{x})$.* ∎

Observe that the clauses (i) and (ii) in Lemma 5.4.2 ensure that $I_\psi^\beta$ is monotonic in $\beta$ and that there exists some ordinal $\kappa$ for which the fixed point is reached. In fact, $I_\psi$ as defined above is obtained after $\kappa$ iterations of $\psi$. Moreover, in this way the least fixed point is obtained indeed. This is an immediate consequence of Lemma 5.4.2(iii).

DEFINITION 5.4.3 (Validity of $\mu p \cdot [\psi(p)]$ and of $\nu p \cdot [\psi(p)]$). Let $\psi$ be a first-order formula over $\mathfrak{M}$, $\psi \in \mathrm{sm}(p)$. We now define

(a) $\mathfrak{M} \models \mu p \cdot [\psi(p)](\bar{x})$ iff $\mathfrak{M} \models I_\psi(\bar{x})$,

(b) $\mathfrak{M} \models \mu p \cdot [\psi(p)]$ iff for all $\bar{x}$, $\mathfrak{M} \models \mu p \cdot [\psi(p)](\bar{x})$, and

(c) $\mathfrak{M} \models \nu p \cdot [\psi(p)]$ iff $\mathfrak{M} \models \neg \mu p \cdot \neg [\psi(p)\{\neg p/p\}]$. ∎

## 5.5. *Acceptable Structures*

As is usual in completeness proofs, we need the ability to encode finite sequences. In our case, this is necessary to define the well-founded set necessarily for applying Orna's rule. For this, we introduce the notion of an acceptable structure (Moschovakis, 1974).[1] First we introduce a number of notions needed for the definition of acceptable structures.

DEFINITION 5.5.1 (Coding scheme, decoding relations, and decoding functions).

(a) A coding scheme for a set $A$ is a triple $\mathscr{C} = \langle N^{\mathscr{C}}, \leqslant^{\mathscr{C}}, \langle\ \rangle^{\mathscr{C}} \rangle$ such that

- (i) $N^{\mathscr{C}} \subseteq A$, $\leqslant^{\mathscr{C}}$ is an ordering on $N^{\mathscr{C}}$ and the structure $\langle N^{\mathscr{C}}, \leqslant^{\mathscr{C}} \rangle$ is isomorphic to the integers with their usual ordering.
- (ii) $\langle\ \rangle^{\mathscr{C}}$ is a one-one function, mapping the set $\bigcup_{n \geqslant 0} A^i$ of all finite sequences over $A$ to $A$. By convention, $A^0 = \phi$; the empty sequence $\langle\ \rangle^{\mathscr{C}}$ is the only sequence of length 0.

(b) With each coding scheme $\mathscr{C}$, we associate the following decoding relations and functions:

- (i) $\mathrm{Seq}^{\mathscr{C}}(x) \Leftrightarrow$ there exist $x_1, ..., x_n$ such that $x = \langle x_1, ..., x_n \rangle^{\mathscr{C}}$. Here, $x = \langle\ \rangle^{\mathscr{C}}$, the code of the empty sequence, is covered by the convention that $x = \langle x_1, ..., x_n \rangle^{\mathscr{C}}$ if $n = 0$.
- (ii) The length function $\mathrm{lh}^{\mathscr{C}}$ for sequences maps $A$ into $N^{\mathscr{C}}$, and

[1] Alternatively, we could have introduced the notion of an arithmetical structure (Harel, 1979).

hence into the integers, because of the isomorphism of $\langle N^{\mathscr{C}}, \leqslant^{\mathscr{C}} \rangle$ with $\langle \mathbb{N}, \leqslant \rangle$:

$$\mathrm{lh}^{\mathscr{C}}(x) = \begin{cases} 0, & \text{if } \neg \mathrm{Seq}^{\mathscr{C}}(x) \\ n, & \text{if } \mathrm{Seq}^{\mathscr{C}}(x) \wedge x = \langle x_1, ..., x_n \rangle^{\mathscr{C}} \text{ for some } x_1, ..., x_n. \end{cases}$$

(iii) The projection $(x)_i^{\mathscr{C}}$, as a function of $x$ and $i$, is defined by

$$(x)_i^{\mathscr{C}} = \begin{cases} x_i, & \text{if } x = \langle x_1, ..., x_n \rangle^{\mathscr{C}} \text{ for some } x_1, ..., x_n, 1 \leqslant i \leqslant n \\ 0, & \text{otherwise.} \end{cases}$$

DEFINITION 5.5.2 (Elementary coding scheme).

(a) A function $f$ is first-order definable on a structure $\mathfrak{M}$ iff its graph is first-order definable, i.e., iff $\{(\bar{x}, \bar{y}) \mid f(\bar{x}) = \bar{y}\}$ is first-order definable on $\mathfrak{M}$.

(b) A coding scheme $\mathscr{C}$ is elementary on a structure $\mathfrak{M}$ if the relations and functions $N^{\mathscr{C}}$, $\leqslant'$, $\mathrm{Seq}^{\mathscr{C}}$, $\mathrm{lh}^{\mathscr{C}}(\ )^{\mathscr{C}}$, are all elementary, i.e., first-order definable on $\mathfrak{M}$.

Note that the class of elementary relations on a structure is closed under conjunction and quantification. This is an immediate consequence of Definition 5.5.2. It follows that the functions $p_n^i$ defined by $p_n^{\mathscr{C}}(x_1, ..., x_n) = \langle x_1, ..., x_n \rangle^{\mathscr{C}}$ are elementary, as $p_n^{\mathscr{C}}(x_1, ..., x_n) = u \Leftrightarrow (\mathrm{Seq}^{\mathscr{C}}(u) \wedge \mathrm{lh}^i(u) = n \wedge \forall i \cdot [1 \leqslant i \leqslant n \supset ((u)_i^{\mathscr{C}} = x_i)])$. (In the sequel, we shall omit the superscripts $\mathscr{C}$.)
As argued before, we need the ability to encode finite sequences. Also fairness arguments require the use of recursive ordinals. In our case these requirements are necessary to define the well-founded set required to apply Orna's rule.

DEFINITION 5.5.3 (Acceptable and ordinal acceptable first-order structures).

(a) A first-order structure $\mathfrak{M}$ is acceptable if there exists a coding scheme elementary on $\mathfrak{M}$.

In the sequel, we consider acceptable structures such that for all recursive ordinals $\alpha$, there exists a constant symbol $\bar{\alpha}$ interpreted as the ordinal $\alpha$. We therefore introduce the notion of an ordinal acceptable structure:

(b) A first-order ordinal acceptable structure is a structure $\mathfrak{M}$ such that:

(i) $\mathfrak{M}$ is an acceptable structure,

(ii) $\mathfrak{M}$'s signature contains symbols $c_i$ for all $i < \omega_1^{ck}$, and $\mathbf{c}_i = i \in |\mathfrak{M}|$, where $\omega_1^{ck}$ is the first non-recursive ordinal, and $\mathbf{c}_i$ denotes the interpretation of $c_i$.

(iii) the predicates Ord ($Ord(a)$ holds iff $a \in |\mathfrak{M}| \cap \omega_1^{ck}$) and $\prec_{\mathrm{Ord}}$, the usual ordering on $\omega_1^{ck}$, are first-order definable in $\mathfrak{M}'$, where $\mathfrak{M}'$ is a reduct on $\mathfrak{M}$, obtained by removing all ordinal constants $c_i$ from its signature.

Let $\mathfrak{M}$ be an ordinal acceptable structure. For completeness, we need amongst others, representability of the guarded commands partial correctness semantics. First note that the I/O-relation of a program $S$ only constrains the valuation of its *free* variables (in the output state). We shall be somewhat more precise below. To do so, suppose that $S$ is a program. Denote by $F$ the set of free variables occurring in $S$. Let $F^c$ denote the complement of $F$, i.e., $F^c$ is the set of all variables not occurring free in $S$. If $R_S^{\mathrm{part}}(\xi, \xi')$ holds, then $R_S^{\mathrm{part}}(\tau, \tau')$ holds, too, provided $\xi | F = \tau | F$, $\xi' | F = \tau' | F$, and $\tau | F^c = \tau' | F^c$, where $|$ denotes restriction. Using this observation, the semantics $R_S^{\mathrm{part}}$, is easily seen to be representable: for example, if $S \equiv {}^*[b \to S']$ then $R_S^{\mathrm{part}}(\xi, \xi') \Leftrightarrow \mathfrak{M} \models \mu X \cdot [(R_b \circ R') \circ X \cup \neg R_b](x, y)$, where $x$ and $y$ are the codes of $\xi | F$ ($\xi' | F$ resp.). Here $R'$ denotes the relation $R_{S'}^{\mathrm{part}}$ associated with $S'$, and $F$ the set of free variables occurring in $S$. Observe that the codes $x$ and $y$ exist since $\mathfrak{M}$ is an ordinal acceptable structure.

We next construct an extension of $\mathfrak{M}$ by adding for every guarded command $S$ a relation symbol $R_S$, interpreted as the semantics $R_S^{\mathrm{part}}$ of $S$. Since $R_S$ is representable, we obtain a structure $\mathfrak{M}'$ such that $\mathrm{Th}(\mathfrak{M}') = \mathrm{Th}(\mathfrak{M})$, where $\mathrm{Th}(\mathfrak{M}) = \{p \in L \mid \mathfrak{M} \models p\}$. I.e., $\mathrm{Th}(\mathfrak{M}')$ is conservative over $\mathrm{Th}(\mathfrak{M})$ and we do not obtain a more expressive language in this way.

We conclude this section by showing that a number of predicates extensively used in the sequel are representable in $L$.

THEOREM 5.5.4 (Representability of a number of predicates). *Assume that $\mathfrak{M}$ is some ordinal acceptable structure. Let $R_1$ and $R_2$ denote binary relations on $|\mathfrak{M}|$ elementary on $\mathfrak{M}$. The following constructs are representable in $L$: $R_1 \circ R_2$, $R_1 \cup R_2$, $R_1^*$, and $\mu p \cdot [R_1 \to p]$.*

*Proof.* It should be clear how to represent $R_1 \circ R_2$ and $R_1 \cup R_2$ in $L$. $R_1^*$ is representable by $\mu X \cdot [R_1 \circ X \cup I]$, where $I$ denotes the identity relation. Finally $\mu p \cdot [R_1 \to p]$ can be represented as follows: define $\phi(x, p) = \forall x' [R_1(x, x') \supset p(x')]$. Then $\mu p \cdot [\phi(x, p)]$ represents $\mu p \cdot [R_1 \to p](x)$. ∎

In the remainder of this paper we shall also use the construct $r \circ R$ for predicates $r$ and binary relations on $|\mathfrak{M}|$, where $\mathfrak{M}$ is as above. Intuitively, $r \circ R$ is satisfied in $x$ iff $x$ is $R$-reachable from some $y$ in which $r$ holds.

DEFINITION 5.5.5 ($r \circ R$). Using the notation as above, we define for predicates $r$ and binary relations $R$ on $|\mathfrak{M}|$ the predicate $r \circ R$ by $r \circ R(x)$

iff $\exists y[r(y) \wedge R(y, x)]$. Observe that $r \circ R$ is trivially representable in $L$, if $R$ is elementary in $\mathfrak{M}$.

In the remainder of this paper $\mathfrak{M}$ always denotes some first-order ordinal acceptable structure.

## 6. Construction of a $\mu$-Term Expressing Strongly Fair Termination

In this section we show that the property "$S$ is strongly fair terminating" is representable in $L$. More precisely, let $*[\Box_{i=1}^{n} b_i \to S_i]$ and let $\mathfrak{M}$ be some ordinal acceptable structure. We construct a formula $\mathrm{SFAIR}(R_1, ..., R_n)$ such that $\mathfrak{M} \models \neg\mathrm{SFAIR}(R_1, ..., R_n)(\xi)$ holds iff "$S$ terminates strongly fair when started in $\xi$." Here, $R_i$ denotes the relation $R_{b_i} \circ R_{S_i}^{sf}$ associated with $b_i$; $S_i$ $(i = 1, ..., n)$.

For programs with two directions, a $\mu$-term expressing strongly fair termination has been constructed in de Roever (1981). To give the reader some intuition, we first construct a term describing the existence of infinite strongly fair execution sequences of a program $S \equiv *[b_1 \to S_1 \Box b_2 \to S_2]$.

From Definition 3.4.2, we obtain that in an infinite strongly fair execution sequence of $S$, either

(1) both directions of $S$ are infinitely often enabled in this sequence, and hence infinitely often taken in it, or

(2) the first direction becomes eventually continuously disabled and the second direction of $S$ is continuously taken from some point onwards in the execution sequence, or

(3) the symmetrical case of (2), i.e., the second direction of $S$ becomes eventually continuously disabled and the first direction is continuously taken from some point onwards in the execution sequence.

The construction of the term describing the existence of an infinite strongly fair execution sequence of $S$ naturally splits up into three cases, according to the three possibilities (1), (2), and (3) above. Let $R_1$ (resp. $R_2$) denote the relations $R_{b_1} \circ R_{S_1}^{sf}$ (resp. $R_{b_2} \circ R_{S_2}^{sf}$) associated with $b_1$; $S_1$ (resp. $b_2$; $S_2$).

*Case* 1. We consider such a sequence as consisting of an infinite number of so-called unconditional fair parts, roughly being finite subsequences of the infinite sequence in which every direction is taken at least once. Such an unconditional part can be described as follows: $(R_1^+ \circ R_2 \cup R_2^+ \circ R_1)$.

This characterization stems from Park (1980). Recall that truth of the predicate $\nu p \cdot [R \circ p]$ in $x_0$ expresses the existence of an infinite sequence $x_0, x_1, x_2, \ldots$ such that $R(x_i, x_{i+1})$ holds for $i \geqslant 0$. As a consequence, the existence of an infinite strongly fair sequence, according to the first possibility above, is captured by the predicate $\nu p \cdot [(R_1^+ \circ R_2 \cup R_2^+ \circ R_1) \circ p]$. This term is called $\mathrm{UF}(R_1, R_2)$.

*Case* 2. We consider possibility (2) above. In this case, the existence of an infinite strongly fair execution sequence of $S$ can be described by a term expressing that after some finite prefix, in which (possibly both) directions 1 and 2 are chosen, only the second direction is continuously taken, since the other one becomes eventually continuously disabled. In the infinite tail of the sequence each intermediate state satisfies $\neg b_1$. This term is captured by $(R_1 \cup R_2)^* \circ \nu p \cdot [((b_2 \wedge \neg b_1) \circ R_2) \circ p]$. This term is called $\mathrm{fair}(R_2)$ $\mathrm{fin}(R_1)$.

*Case* 3. Symmetrically to case (2) the existence of such an execution sequence can be described by $\mathrm{fair}(R_1)\,\mathrm{fin}(R_2)$.

Now define $\mathrm{SFAIR}(R_1, R_2)$ by $\mathrm{SFAIR}(R_1, R_2) = \mathrm{UF}(R_1, R_2) \vee \mathrm{fair}(R_2)$ $\mathrm{fin}(R_1) \vee \mathrm{fair}(R_1)\,\mathrm{fin}(R_2)$. We then obtain that $S$ admits an infinite strongly fair execution sequence iff $\mathrm{SFAIR}(R_1, R_2)$ holds.

The structure of Section 6 is as follows: in Section 6.1 we describe the predicate $\mathrm{UF}(R_1, \ldots, R_n)$ for $n \geqslant 1$. This predicate is a generalization of $\mathrm{UF}(R_1, R_2)$ that we derived in case (1) above. In Section 6.2 we extend the reasoning of case (2), hence case (3), when there are more than two directions in a repetition. Finally, in Section 6.3 we show that for every command $S$ and command $q$, the weakest precondition for fair termination is definable in $L$.

### 6.1. *Unconditionally Fair Termination*

At first, we consider execution sequences of programs $*[\Box_{i=1}^{n} b_i \rightarrow S_i]$, in which each direction of $S$ is chosen infinitely often. Any such sequence is strongly fair iff it is unconditionally fair. In the sequel, we assume that $R_1, \ldots, R_n$ are the relations $R_{b_1} \circ R_1^{\mathrm{sf}}, \ldots, R_{b_n} \circ R_n^{\mathrm{sf}}$ associated with the statements $b_1; S_1, \ldots, b_n; S_n$. Consequently, we first consider the problem of describing in $L$ the existence of an infinite sequence of $R_i$-moves in which each of the $R_i$ occurs infinitely often ($i = 1, \ldots, n$).

Consider such an infinite sequence. Since each $R_i$ ($i = 1, \ldots, n$) occurs an infinite number of times, this sequence may be viewed as consisting of an infinite number of finite sequences, the so-called *U(nconditional)parts*. Every *Upart* satisfies:

(i) each $R_i$ occurs in the Upart.

(ii) this Upart is the smallest sequence satisfying (i); i.e., any initial fragment of Upart leaves some $R_i$ out.

To define a relation Upart($R_1, ..., R_n$), which expresses for every pair of states $(\xi, \xi')$, whether $\xi'$ can be reached from $\xi$ by executing an Upart (w.r.t. $R_1, ..., R_n$), it suffices to consider Uparts in which the first occurrences of the moves are in some predescribed order, so-called *Usegments*, since any Upart of $R_1, ..., R_n$ is an Usegment of some permutation $R_{i_1}, ..., R_{i_n}$. More clearly, a Usegment of the ordered sequence of moves $R_1, ..., R_n$ is a finite sequence in which for no $i, j$ with $1 \leqslant i < j \leqslant n$ a $R_j$-move occurs before a $R_i$-move has occurred.

The relation Usegment($R_1, ..., R_n$) is defined inductively (w.r.t. $n$) as follows: The case $n = 1$ is simple: define Usegment($R_1$) = $R_1$.

Now, suppose that Usegment($R_1, ..., R_k$) has been defined. Then, Usegment($R_1, ..., R_{k+1}$) looks like $R_1, ..., R_i, ..., R_k, ..., R_{k+1}$, where the first occurrences of $R_1$, $R_i$, $R_k$, $R_{k+1}$ are shown $(1 < i < k)$. First, observe that $R_{k+1}$ occurs only once; this is a consequence of requirement (ii) above. Second, observe that the prefix $R_1, ..., R_i, ..., R_k$ of the above sequence is a Usegment of $R_1, ..., R_k$. Hence, the sequence up to, but not including $R_{k+1}$ is not necessarily a Upart of $R_1, ..., R_k$. However, it starts at least with a Usegment of $R_1, ..., R_k$. The remaining part may contain any (finite) number of $R_i$-occurrences (but no $R_{k+1}$). This motivates the following definitions.

DEFINITION 6.1.1 (Usegment($R_1, ..., R_n$) for $n \geqslant 1$). Usegment($R_1$) = $R_1$ and for $n \geqslant 1$:

$$\text{Usegment}(R_1, ..., R_{n+1}) = \text{Usegment}(R_1, ..., R_n) \circ (R_1 \cup \cdots \cup R_n)^* \circ R_{n+1}.$$

EXAMPLE. $\text{Usegment}(R_1, R_2, R_3) = R_1 \circ R_1^* \circ R_2 \circ (R_1 \cup R_2)^* \circ R_3$.

DEFINITION 6.1.2 (Upart($R_1, ..., R_n$) for $n \geqslant 1$). For $n \geqslant 1$: Upart($R_1, ..., R_n$) $= \bigcup_{i_1, ..., i_n \text{ perm of } 1, ..., n} \text{Usegment}(R_{i_1}, ..., R_{i_n})$. I.e., in Upart($R_1, ..., R_n$) the order of the $R_i$ $(i = 1, ..., n)$ is immaterial.

Remembering the example given above, the existence of an infinite sequence of Uparts, starting in a state $\xi$, is expressed by satisfaction of a predicate UF($R_1, ..., R_n$) in $\xi$, defined as follows:

DEFINITION 6.1.3 (UF($R_1, ..., R_n$) for $n \geqslant 1$). For $n \geqslant 1$: UF($R_1, ..., R_n$) $= \nu p \cdot [\text{Upart}(R_1, ..., R_n) \circ p]$. (Recall that $R_i$ denote relations.)

An execution sequence of a program $S \equiv {}^*[\square_{i=1}^n b_i \rightarrow S_i]$ in which each direction is chosen infinitely often is strongly fair iff it is unconditionally fair. Consequently, the program $S \equiv {}^*[\square_{i=1}^n b_i \rightarrow S_i]$ $(n \geqslant 1)$ admits an infinite unconditionally fair execution sequence starting in $\xi$ iff UF($R_1, ..., R_n$) holds in $\xi$. Recall that $R_i$ denotes the relation $R_{b_i} \circ R_{S_i}^{sf}$ associated with $b_i$; $S_i$ $(i = 1, ..., n)$.

6.2. *Strongly Fair Termination*

Now, consider infinite sequences of a program $*[\Box_{i=1}^{n} b_i \rightarrow S_i]$ in which directions can become disabled. Suppose that the $n$th direction $b_n; S_n$ becomes *eventually never enabled any more.* Then an infinite strongly fair sequence of $R_1, ..., R_n$-moves consists of some finite sequence of $R_1, ..., R_n$-moves followed by an infinite strongly fair sequence of $R_1, ..., R_{n-1}$-moves in which every intermediate state satisfies $\neg b_n$. In case no other direction of $S$ becomes eventually continuously disabled, the existence of such a sequence is expressed by a predicate $(R_1 \cup \cdots \cup R_n)^* \circ \mathrm{UF}(\neg b_n \circ R_1, ..., \neg b_n \circ R_{n-1})$. Observe that this predicate is equivalent to $(b_1 \circ R_1 \cup \cdots \cup b_n \circ R_n)^* \circ \mathrm{UF}((b_1 \wedge \neg b_n) \circ R_1, ..., (b_{n-1} \wedge \neg b_n) \circ R_{n-1})$, since the enabling condition $b_i$ is incorporated in $R_i$ ($i = 1, ..., n$). The possibility that other moves may become disabled, too, leads to the following definition[2]:

DEFINITION 6.2.1 ($\mathrm{fair}(b_{i_1} \circ R_{i_1}, ..., b_{i_k} \circ R_{i_k})\, \mathrm{fin}(b_{i_{k+1}} \circ R_{i_{k+1}}, ..., b_{i_n} \circ R_{i_n})$ for $n \geq 2$ and $1 \leq k < n$). Let $n \geq 2$ and suppose that $i_1, ..., i_n$ is some permutation of $1, ..., n$. For $k$, satisfying $1 \leq k < n$, define

$$\begin{aligned}
&\mathrm{fair}(b_{i_1} \circ R_{i_1}, ..., b_{i_k} \circ R_{i_k})\, \mathrm{fin}(b_{i_{k+1}} \circ R_{i_{k+1}}, ..., b_{i_n} \circ R_{i_n}) \\
&\quad = \left( \bigcup_{i=1}^{n} b_i \circ R_i \right)^* \circ \mathrm{UF}\left( \left( b_{i_1} \wedge \bigwedge_{j=k+1}^{n} \neg b_{i_j} \right) \circ R_{i_1}, ..., \right. \\
&\qquad\qquad \left. \left( b_{i_k} \wedge \bigwedge_{j=k+1}^{n} \neg b_{i_j} \right) \circ R_{i_k} \right).
\end{aligned}$$

*Remark.* $\mathrm{fair}(b_{i_1} \circ R_{i_1}, ..., b_{i_k} \circ R_{i_k})\, \mathrm{fin}(b_{i_{k+1}} \circ R_{i_{k+1}}, ..., b_{i_n} \circ R_{i_n})$ holds in state $\xi$ iff there exists an infinite strongly fair sequence, starting in $\xi$, in which the directions $b_{i_{k+1}}; S_{i_{k+1}}, ..., b_{i_n}; S_{i_n}$ are eventually never enabled any more.

Now, finally the predicate expressing the existence of infinite strongly fair sequences can be formulated.

DEFINITION 6.2.2 ($\mathrm{SFAIR}(b_1 \circ R_1, ..., b_n \circ R_n)$ for $n \geq 1$). $\mathrm{SFAIR}(b_1 \circ R_1) = \mathrm{UF}(b_1 \circ R_1)$, and for $n \geq 2$:

$$\begin{aligned}
&\mathrm{SFAIR}(b_1 \circ R_1, ..., b_n \circ R_n) \\
&\quad = \mathrm{UF}(b_1 \circ R_1, ..., b_n \circ R_n) \vee \bigvee_{\substack{i_1, ..., i_n \text{ perm of } 1, ..., n \\ 1 \leq k < n}} \mathrm{fair}(b_{i_1} \circ R_{i_1}, ..., b_{i_k} \circ R_{i_k}) \\
&\qquad\qquad \mathrm{fin}(b_{i_{k+1}} \circ R_{i_{k+1}}, ..., b_{i_n} \circ R_{i_n}).
\end{aligned}$$

[2] This definition is due to P. van Emde Boas.

In the sequel we always assume that the relation $b_i$ is incorporated in the relation $R_i$. Also, with $R_i$ we always associate $b_i$ as enabling condition. Thus, $R_i$ will denote the relation $R_{b_i} \circ R^{sf}_{S_i}$.

We defined here, for every sequence of relations $R_1, ..., R_n$ a *different* predicate. In other words, SFAIR is not a second order formula! For the proof of Theorem 6.3.4 we need the following technical lemma.

LEMMA 6.2.3 (Characterization of SFAIR$(R_1, ..., R_n)$).

$$\mathfrak{M} \models \neg \mathrm{SFAIR}(R_1, ..., R_n)$$
$$\Leftrightarrow \Bigg[ \mathfrak{M} \models \neg \mathrm{UF}(R_1, ..., R_n) \wedge \bigwedge_{\substack{i_1, ..., i_n \text{ perm of } 1, ..., n \\ 1 \leqslant k < n}} \left( \bigcup_{i=1}^{n} R_i \right)^*$$
$$\rightarrow \neg \mathrm{UF} \left( \bigwedge_{j=k+1}^{n} \neg b_{i_j} \circ R_{i_1}, ..., \bigwedge_{j=k+1}^{n} \neg b_{i_j} \circ R_{i_k} \right) \Bigg].$$

*Proof.* For $n = 1$ this follows by Definition 6.2.2. So assume that $n \geqslant 2$. Then the lemma follows from Definition 6.2.2, Definition 6.2.1, and Lemma 5.1.3. ∎

### 6.3. *Weakest Precondition for Strongly Fair Termination*

As a last preparation for the soundness and completeness proofs, we mention the notions of the weakest liberal precondition and of the weakest precondition for strongly fair termination.

DEFINITION 6.3.1 (Weakest liberal precondition). An assertion $p = \mathrm{wlp}(S, q)$ is the weakest liberal precondition w.r.t. a command $S$ and a condition $q$ if $\mathfrak{M} \models [p]\ S[q]_{\mathrm{part}}$ and for each $r$, $\mathfrak{M} \models [r]\ S[q]_{\mathrm{part}}$ implies $\mathfrak{M} \models r \supset p$.

In (de Bakker, 1980), it has been shown that for each command $S$ and assertion $q$, $\mathrm{wlp}(S, q)$ is definable in $L$. It is useful to mention that for loops $S$ $\mathrm{wlp}(S, q) = (((\bigcup_{i=1}^{n} R_i)^* \circ \bigwedge_{i=1}^{n} \neg b_i) \rightarrow q)$.

DEFINITION 6.3.2 (Weakest precondition for strongly fair termination). An assertion $p$ is the weakest precondition for strongly fair termination w.r.t. a command $S$ and a condition $q$ if $\mathfrak{M} \models [p]\ S[q]_{sf}$ and for each $r$, $\mathfrak{M} \models [r]\ S[q]_{sf}$ implies $\mathfrak{M} \models r \supset p$.

We next state the key result of this section, viz., the definability of the weakest precondition for strongly fair termination sfwp$(S, q)$ for any command $S$ and any condition $q$. In Theorem 6.3.4 below, we prove that wpsf indeed defines the weakest precondition for strongly fair termination.

DEFINITION 6.3.3 (sfwp$(S, q)$). For each command $S$ and condition $q$, sfwp$(S, q)$ is inductively defined by

(a) sfwp$(x := e, q) = q\{e/x\}$,

(b) sfwp$(S_1 ; S_2, q) =$ sfwp$(S_1,$ sfwp$(S_2, q))$, where $S_1$ and $S_2$ are simple commands, and

(c) sfwp$(*[\Box_{i=1}^n b_i \rightarrow S_i], q) = \neg \mathrm{SFAIR}(R_1, ..., R_n) \wedge (((\bigcup_{i=1}^n R_i)^* \circ \bigwedge_{i=1}^n \neg b_i) \rightarrow q)$, where $S_i$ are assumed to be simple.

THEOREM 6.3.4. *For each command $S$ and condition $q$, sfwp$(S, q)$ is indeed the weakest precondition for strongly fair termination w.r.t. $S$ and $q$.*

*Proof.* The proof is standard except for the case that $S \equiv *[\Box_{i=1}^n b_i \rightarrow S_i]$ with simple $S_i$, $i = 1, ..., n$. Consequently, we prove that both

(a) $\mathfrak{M} \models [\mathrm{sfwp}(*[\Box_{i=1}^n b_i \rightarrow S_i], q)]^* [\Box_{i=1}^n b_i \rightarrow S_i][q]_{sf}$, and

(b) $\mathfrak{M} \models [r]^*[\Box_{i=1}^n b_i \rightarrow S_i][q]_{sf} \Rightarrow \mathfrak{M} \models r \supset \mathrm{sfwp}(*[\Box_{i=1}^n b_i \rightarrow S_i], q)$ hold.

To do so, it suffices to prove that for every $\xi$: $\mathfrak{M} \models [r]\ *[\Box_{i=1}^n b_i \rightarrow S_i]$ $[q](\xi) \Leftrightarrow \mathfrak{M} \models r \supset (\neg \mathrm{SFAIR}(R_1, ..., R_n) \wedge (((\bigcup_{i=1}^n R_i)^* \circ \bigwedge_{i=1}^n \neg b_i) \rightarrow q))$ $(\xi)$, holds.

($\Rightarrow$) Suppose that $\mathfrak{M} \models [r]\ *[\Box_{i=1}^n b_i \rightarrow S_i][q]_{sf}$ holds. Choose some state $\xi$ such that $\mathfrak{M} \models r(\xi)$ holds. Assume, to obtain a contradiction, that $\mathfrak{M} \models \mathrm{SFAIR}(R_1, ..., R_n)(\xi)$. Then this leads immediately to a contradiction, since this implies the existence of an infinite strongly fair execution sequence, starting in $\xi$. So $\mathfrak{M} \models \neg \mathrm{SFAIR}(R_1, ..., R_n)(\xi)$ holds. It remains to prove that $\mathfrak{M} \models (((\bigcup_{i=1}^n R_i)^* \circ \bigwedge_{i=1}^n \neg b_i) \rightarrow q)(\xi)$ holds, too. To do this, choose some $\xi'$ satisfying $\mathfrak{M} \models ((\bigcup_{i=1}^n R_i)^* \circ \bigwedge_{i=1}^n \neg b_i)(\xi, \xi')$. Clearly, then also $\mathfrak{M} \models R_S^{sf}(\xi, \xi')$, where $S \equiv *[\Box_{i=1}^n b_i \rightarrow S_i]$, and so by the hypothesis $\mathfrak{M} \models q(\xi')$.

($\Leftarrow$) Suppose that $\mathfrak{M} \models r \supset (\neg \mathrm{SFAIR}(R_1, ..., R_n) \wedge ((\bigcup_{i=1}^n R_i)^* \circ \bigwedge_{i=1}^n \neg b_i) \rightarrow q)$. Choose state $\xi$ such that $\mathfrak{M} \models r(\xi)$. Since, by hypothesis $\mathfrak{M} \models \neg \mathrm{SFAIR}(R_1, ..., R_n)(\xi)$, the repetition always terminates strongly fair. We have to prove that, in this case, each final state satisfies $q$. Choose some $\xi'$ such that $\mathfrak{M} \models R_S^{sf}(\xi, \xi')$, where $S \equiv *[\Box_{i=1}^n b_i \rightarrow S_i]$. Clearly, then also $\mathfrak{M} \models ((\bigcup_{i=1}^n R_i)^* \circ \bigwedge_{i=1}^n \neg b_i)(\xi, \xi')$ and so, by the hypothesis, $\mathfrak{M} \models q(\xi')$ holds, which had to be shown. ∎

COROLLARY 6.3.5. *For every $\xi$: $\mathfrak{M} \models$ sfwp$(*[\Box_{i=1}^n b_i \rightarrow S_i], true)(\xi) \Leftrightarrow$ $\mathfrak{M} \models \neg \mathrm{SFAIR}(R_1, ..., R_n)(\xi)$.*

This corollary states that strongly fair termination of a repetition is indeed expressible in the $\mu$-calculus.

## 7. COMPLETENESS

In this section, we prove the completeness of our proof system, i.e., we will show that for any statement $S \in \mathrm{LGC}(\mathfrak{M})$, assertions $r, q \in L$,

$$\mathfrak{M} \models [r]\, S[q]_{st} \Rightarrow \mathrm{Th}(\mathfrak{M}) \vdash [r]\, S[q] \qquad \text{holds.} \qquad (*)$$

Here $\mathfrak{M}$ is by convention a first-order ordinal acceptable structure, and $\mathrm{Th}(\mathfrak{M}) = \{p \in L \mid \mathfrak{M} \models p\}$. As is usual in such proofs, completeness is established by structural induction on the complexity of statements $S$. Observe that $(*)$ is trivial in case $S$ is not a repetition. Therefore to prove $(*)$, it suffices to concentrate on the case where $S \equiv {}^*[\square_{i=1}^n b_i \to S_i]$ with $n \geqslant 1$. In this case, we establish $(*)$ by induction on $n$, the number of directions in $S$. Next observe that when $n = 1$ the proof of $(*)$ is straightforward. Consequently, we proceed with loops with more than one direction, the induction hypothesis being

INDUCTION HYPOTHESIS (IH). (a) *and* (b) *below both hold*:

(a) *for all simple commands* $S$, $\mathfrak{M} \models [r]\, S[q]_{st} \Rightarrow \mathrm{Th}(\mathfrak{M}) \vdash [r]\, S[q]$.

(b) *for all* $k$, $1 \leqslant k < n$, $\mathfrak{M} \models [r]\ {}^*[\square_{i=1}^k b_i \to S_i][q]_{st} \Rightarrow \mathrm{Th}(\mathfrak{M}) \vdash [r]\ {}^*[\square_{i=1}^k b_i \to S_i][q]$.

From the discussion above it follows that we may assume that $S$ is a repetition with at least two directions and that (IH) holds. Consequently, we are going to prove that given the fact that $\mathfrak{M} \models [r]\ {}^*[\square_{i=1}^n b_i \to S_i][q]_{st}$ holds for $n \geqslant 2$, we can define in $L$ the auxiliary quantities, i.e., a well-founded set $(W, \prec)$, a ranking predicate $\pi$, and pairwise disjoint sets $D_w$ and $St_w$ for $w \in W$, $w \succ 0$, such that the premisses (a), (b), (c), and (d) of Orna's rule as stated in Section 1 hold. The definitions of the auxiliary quantities are developed in Section 7.1. In Lemmata 7.2.1 through 7.2.4, validity of premisses (a) through (d) are proved, culminating in completeness theorem 7.2.5, whose proof is then standard.

### 7.1. *The Auxiliary Quantities*

Assume that $\mathfrak{M} \models [r]\ {}^*[\square_{i=1}^n b_i \to S_i][q]_{st}$ holds. The main results of this section are that the auxiliary quantities necessary to apply Orna's rule are definable within $L$.

First we are going to define a well-founded set $W$ and a predicate $\pi\colon W \to (\text{States} \to \{\mathit{true}, \mathit{false}\})$, ranking every state (reachable by $S$). To do so, we observe that the usual approach of counting moves does not work, because not every move brings the program closer to termination.

E.g., in case of Dijkstra's random number generator, see Section 3.4, move $R_1$ will not help reach termination.

Now $S$ terminates strongly fair and hence also unconditionally fair. This follows from Theorem 3.4.4. At any time, there is at least one decreasing move; otherwise there exists a state in which no move would bring the program closer to termination, resulting in the existence of an infinite strongly fair sequence, yielding a contradiction. So, if in a successive sequence of iterations, "every enabled move has been executed at least once," then certainly the program has come closer to termination. This shows that viewing execution sequences as consisting of Uparts is a natural thing to do. Unfortunately, counting Uparts does not quite work, because we have to rank *all* states in order for Orna's rule to apply.

Consider such a Upart. It suffices that the states reached by executing this Upart, are ranked in such a way that it reflects the "progress" that is made w.r.t. executing this Upart itself. Now a move leads to "progress" if it is a new one that has not been made in the Upart as yet. This gives the intuition behind the definitions of $W$ and $\pi$ that we now develop. First, we consider the problem of ranking states related by Uparts in more detail. At this stage, we therefore disregard the internal progress within a Upart; such progress is incorporated afterwards.

Consider any reachable state $\xi$. Intuitively this state will be ranked by counting the number of Uparts necessary to reach a final state, i.e., $\xi$ will be ranked by $\beta$ if it takes the program at most $\beta$ Uparts from $\xi$ to reach termination. To define the rank $\beta$ of $\xi$, we apply the techniques developed in Section 5. Define $\tau(p) = \lambda\xi \cdot (\text{Upart}(R_1, ..., R_n) \to p)(\xi)$. From Lemma 5.1.2 it follows that the least fixed point of $\tau$ exists and that it can be obtained by iteration. Intuitively, $\tau^{\beta}(\mathit{false})$ holds in $\xi$ if in $\xi$ we are at most $\beta$ Uparts away from termination. It also follows from Lemma 5.1.2 that there exists some $\lambda$ such that

$$\tau^{\lambda}(\mathit{false}) = \mu p \cdot [\text{Upart}(R_1, ..., R_n) \to p] \qquad \text{holds.} \tag{A}$$

Let $\bar{\alpha}$ be the least ordinal satisfying (A). $\bar{\alpha}$ is a recursive ordinal, cf. Apt and Plotkin (1985). Therefore, we have that for all $\beta \leqslant \bar{\alpha}$, $\beta$ is a recursive ordinal, too.

Of course, for this idea to work we need to show that $\tau^{\beta}(\mathit{false})$ is representable by a formula in $L$.

7.1.1. Theorem (Definability of $\tau^{\beta}$(false)). *Let $\tau(p) = \lambda\xi \cdot (\text{Upart}(R_1, ..., R_n) \to p)(\xi)$. There exists a formula $\phi$ in L such that for all $\xi$ and all $\beta \leqslant \bar{\alpha} \cdot \tau^{\beta}(\mathit{false})(\xi)$ holds iff $\mathfrak{M} \models \phi(\beta)(\xi)$.*

*Proof.* Define $\phi(\beta) = \mu r \cdot [\exists \alpha < \beta \cdot (\text{Upart}(R_1, ..., R_n) \to r(\alpha))]$. By induction on $\beta \leqslant \bar{\alpha}$ we prove that for all $\beta \leqslant \bar{\alpha}$ and all $\xi$, $\tau^{\beta}(\mathit{false})(\xi)$ holds iff $\mathfrak{M} \models \phi(\beta)(\xi)$.

Induction basis, $\beta = \bar{0}$. Trivial, since for all $\xi$, $\tau^0(\mathit{false})(\xi) \Leftrightarrow$ false and $\mathfrak{M} \models \phi(\bar{0})(\xi) \Leftrightarrow \mathfrak{M} \models \mathit{false}(\xi) \Leftrightarrow$ false.

Induction hypothesis (IH). For all $\lambda < \beta$ and all $\xi$, $\tau^\lambda(\mathit{false})(\xi)$ holds iff $\mathfrak{M} \models \phi(\lambda)(\xi)$.

Induction step. For $\beta = \bar{0}$, we have that

$$\mathfrak{M} \models \phi(\beta)(\xi) \Leftrightarrow \mathfrak{M} \models \mu r \cdot [\exists \alpha < \beta \cdot (\mathrm{Upart}(R_1, ..., R_n) \to r(\alpha))](\xi)$$

(definition of $\phi$)

$$\Leftrightarrow \mathfrak{M} \models \exists \alpha < \beta \cdot (\mathrm{Upart}(R_1, ..., R_n) \to \phi(\alpha))(\xi)$$

(fixed point property)

$$\Leftrightarrow \text{for some } \lambda < \beta, \mathfrak{M} \models (\mathrm{Upart}(R_1, ..., R_n) \to \phi(\lambda))(\xi)$$

$$\Leftrightarrow \text{for some } \lambda < \beta \text{ and for all } \xi',$$

$$\mathfrak{M} \models [\mathrm{Upart}(R_1, ..., R_n)(\xi, \xi') \supset \phi(\lambda)(\xi')]$$

$$\Leftrightarrow \text{for some } \lambda < \beta \text{ and for all } \xi',$$

$$\mathfrak{M} \models [\mathrm{Upart}(R_1, ..., R_n)(\xi, \xi')] \Rightarrow \tau^\lambda(\mathit{false})(\xi') \text{ (IH)}$$

$$\Leftrightarrow \text{for all } \xi', \mathfrak{M} \models \mathrm{Upart}(R_1, ..., R_n)(\xi, \xi') \Rightarrow (\exists \lambda < \beta \cdot \tau^\lambda(\mathit{false})(\xi'))$$

$$\Leftrightarrow \text{for all } \xi', \mathfrak{M} \models \mathrm{Upart}(R_1, ..., R_n)(\xi, \xi') \Rightarrow \bigsqcup_{\lambda < \beta} \tau^\lambda(\mathit{false})(\xi')$$

$$\Leftrightarrow \tau^\beta(\mathit{false})(\xi). \quad \blacksquare$$

Now, we define the well-founded ordered set $W$: each $w \in W$, $w$ not minimal, consists of two components. The first one counts Uparts, the second one records "progress" within the last (incomplete) Upart and is a sequence of length at most $n$, the number of directions within this Upart, which records the directions within this Upart, that have already been taken.

We next define the predicate $\mathrm{seq}_n(s)$ which holds iff $s$ is sequence of length at most $n$, in which directions are recorded only and in which each direction is recorded at most once.

Definition 7.1.2 ($\mathrm{seq}_n$).

$$\mathrm{seq}_n(s) = \mathrm{Seq}(s) \wedge \mathrm{lh}(s) \leqslant n \wedge \forall i[(1 \leqslant i \leqslant \mathrm{lh}(s)) \supset (1 \leqslant (s)_i \leqslant n)]$$
$$\wedge \forall i, j[(1 \leqslant i, j \leqslant \mathrm{lh}(s) \wedge i \neq j) \supset (s)_i \neq (s)_j]$$

(cf. Definition 5.5.1).

Next, we define the well-founded structure required to apply Orna's rule.

DEFINITION 7.1.3 (The well-founded structure $W_{\alpha,n}$).

(a) $W_{\bar{\alpha},n} = \{(\bar{\lambda}, s) \mid \bar{0} \leqslant \bar{\lambda} \leqslant \bar{\alpha} \wedge \mathrm{seq}_n(s)\} \cup \{\bar{0}\}$.

(b) The ordering $\prec$ defined on $W_{\bar{\alpha},n}$ is the following: $\bar{0} \prec (\bar{\lambda}, s)$ for all $(\bar{\lambda}, s) \in W_{\bar{\alpha},n}$, and $(\bar{\lambda}_1, s_1) \prec (\bar{\lambda}_2, s_2)$ iff $(\bar{\lambda}_1 < \bar{\lambda}_2) \vee ((\bar{\lambda}_1 = \bar{\lambda}_2) \wedge \mathrm{lh}(s_2) < \mathrm{lh}(s_1) \wedge \forall i[(1 \leqslant i \leqslant \mathrm{lh}(s_2)) \supset (s_2)_i = (s_1)_i])$.

Next, we define the ranking predicate $\pi$.

DEFINITION 7.1.4 (The ranking predicate $\pi$). The predicate $\pi$: $W_{\bar{\alpha},n} \rightarrow$ (States $\rightarrow \{\mathit{true}, \mathit{false}\}$) is defined by:

$$\pi(\bar{\lambda}, \langle\ \rangle) = \tau^{\bar{\lambda}}(\mathit{false}) \wedge r \circ \left(\bigcup_{i=1}^{n} R_i\right)^* \wedge \bigvee_{i=1}^{n} b_i,$$

$$\pi(\bar{\lambda}, \langle i_1, ..., i_k\rangle) = \tau^{\bar{\lambda}}(\mathit{false}) \circ \left(\mathrm{Usegment}(R_{i_1}, ..., R_{i_k}) \circ \left(\bigcup_{j=1}^{k} R_{i_j}\right)^*\right)$$
$$\wedge\, r \circ \left(\bigcup_{i=1}^{n} R_i\right)^* \wedge \bigvee_{i=1}^{n} b_i \qquad (\text{for } 1 \leqslant k < n),$$

$$\pi(\bar{\lambda}, \langle i_1, ..., i_n\rangle) = \bigsqcup_{\beta < \lambda} \tau^{\beta}(\mathit{false}) \wedge r \circ \left(\bigcup_{i=1}^{n} R_i\right)^* \wedge \bigvee_{i=1}^{n} b_i,$$

$$\pi(\bar{0}) = r \circ \left(\bigcup_{i=1}^{n} R_i\right)^* \wedge \bigwedge_{i=1}^{n} \neg b_i.$$

Note that accessibility is demanded for $\pi(w)$, $w \in W_{\bar{\alpha},n}$. If $1 \leqslant k < n$ and $\pi(\bar{\lambda}, \langle i_1, ..., i_k\rangle)(\xi)$ holds, then there exists a state $\xi'$ in which the program is at most $\bar{\lambda}$ Uparts away from termination. It takes a fragment, i.e., an initial part of a Usegment to reach $\xi$ from $\xi'$, namely $\mathrm{Usegment}(R_{i_1}, ..., R_{i_k}) \circ (\bigcup_{j=1}^{k} R_{i_j})^*$.

Defining $St_w$ and $D_w$ for $w \succ \bar{0}$, $w \in W_{\bar{\alpha},n}$, is simple now. If we are at the start of a Upart, i.e., $w = (\bar{\lambda}, \langle\ \rangle)$ or $w = (\bar{\lambda}, \langle i_1, ..., i_n\rangle)$ for some $\bar{\lambda} \leqslant \bar{\alpha}$, then every move leads to eventual completion of this Upart. Otherwise, $w = (\bar{\lambda}, \langle i_1, ..., i_k\rangle)$ for some $\bar{\lambda}$, $1 \leqslant k < n$, and only moves different from $R_{i_1}, ..., R_{i_k}$ lead to eventual completion of this Upart.

7.1.5. DEFINITION (The set of helpful and steady moves $D_w$ and $St_w$). Let $w \in W_{\bar{\alpha},n}$, $w \succ 0$. Then $w = (\bar{\lambda}, s)$ for some $\bar{\lambda} \leqslant \bar{\alpha}$, and $s$ with $\mathrm{seq}_n(s)$.

If $\mathrm{lh}(s) = 0$ or if $\mathrm{lh}(s) = n$, then $D_w = \{1, ..., n\}$ and $St_w = \emptyset$.

If $0 < \mathrm{lh}(s) < n$, then $D_w = \{i \mid (1 \leqslant i \leqslant n) \wedge \forall j \cdot 1 \leqslant j \leqslant \mathrm{lh}(s)[(s)_j \neq i]\}$, $St_w = \{1, ..., n\} - D_w$.

Note that for all $w \in W_{\alpha,n}$, $w \succ \bar{0}$: $D_w \cap St_w = \varnothing$, $D_w \neq \varnothing$ and $D_w \cup St_w = \{1, ..., n\}$.

### 7.2. *Completeness of Orna's Rule*

Using the above definitions, we next prove that the four premises, (a)–(d) of Orna's rule are valid. To be more precise, Lemmata 7.2.1–7.2.4 below show that these four premises are satisfied indeed. From the induction hypothesis, completeness of the rule and hence of our proof system then easily follows. Assume that $\mathfrak{M} \models [r]\ {}^*[\Box_{i=1}^{n}\ b_i \to S_i][q]_{sf}$ holds.

By Definition 6.3.3 and Theorem 6.3.4 we may assume that $\mathfrak{M} \models r \supset (\neg \mathrm{SFAIR}(R_1, ..., R_n) \wedge (((\bigcup_{i=1}^{n} R_i)^* \circ \bigwedge_{i=1}^{n} \neg b_i) \to q))$ holds, too.

LEMMA 7.2.1 (Corresponding to premise (a) of Orna's rule). *Let $w \in W_{\alpha,n}$, $j \in D_w$; i.e., $R_j$ is a decreasing move. Suppose that $\mathfrak{M} \models r \supset (\neg \mathrm{SFAIR}(R_1, ..., R_n) \wedge ((\bigcup_{i=1}^{n} R_i)^* \circ \bigwedge_{i=1}^{n} \neg b_i) \to q)$ holds. Then $\mathfrak{M} \models [\pi(w) \wedge w \succ \bar{0} \wedge b_j]\ S_j [\exists v \prec w \cdot \pi(v)]$ holds, too.*

*Proof.* We have to prove that for all $\xi, \xi' \in$ States such that $\mathfrak{M} \models R_j(\xi, \xi')$, $\mathfrak{M} \models (\pi(w) \wedge w \succ \bar{0})(\xi) \Rightarrow \mathfrak{M} \models \exists v \prec w \cdot \pi(v)(\xi')$.[3] Choose states $\xi$ and $\xi'$ satisfying $\mathfrak{M} \models R_j(\xi, \xi')$ and suppose that $\mathfrak{M} \models (\pi(w) \wedge w \succ \bar{0})(\xi)$ holds. To prove the lemma, we distinguish two cases:

(a) $\mathfrak{M} \models \bigwedge_{i=1}^{n} \neg b_i(\xi')$. In this case, $\mathfrak{M} \models \pi(\bar{0})(\xi')$, and we are done.

(b) $$\mathfrak{M} \models \bigvee_{i=1}^{n} b_i(\xi'). \tag{i}$$

ince $\mathfrak{M} \models \pi(w)(\xi)$ holds, $\mathfrak{M} \models r \circ (\bigcup_{i=1}^{n} R_i)^*\ (\xi)$ holds, too. I.e.,

$$\mathfrak{M} \models \exists \xi'' \left[ r(\xi'') \wedge \left( \bigcup_{i=1}^{n} R_i \right)^* (\xi'', \xi) \right] \qquad \text{holds.} \tag{ii}$$

s a consequence of Fact 5.2.5, we obtain that $(\bigcup_{i=1}^{n} R_i)^* \circ R_j \subseteq$ $\bigcup_{i=1}^{n} R_i)^*$. Therefore, it follows from $\mathfrak{M} \models R_j(\xi, \xi')$ and (ii) that $\mathfrak{M} \models$ $''[r(\xi'') \wedge (\bigcup_{i=1}^{n} R_i)^*\ (\xi'', \xi')]$ holds, too; i.e.,

$$\mathfrak{M} \models r \circ \left( \bigcup_{i=1}^{n} R_i \right)^* (\xi'). \tag{iii}$$

xt, let $w = (\lambda, s)$. We are going to prove that $\mathfrak{M} \models \exists v \prec w \cdot \pi(v)(\xi')$ lds. To do so, we distinguish three cases:

(1) $\mathrm{lh}(s) = 0$, i.e., $s = \langle\ \rangle$. Since $\mathfrak{M} \models \pi(w)(\xi)$, $\mathfrak{M} \models \tau^{\lambda}(\mathit{false})(\xi)$ 'ds. Consequently, it follows that $\mathfrak{M} \models \exists \xi''[\tau^{\lambda}(\mathit{false})(\xi'') \wedge R_j(\xi'', \xi')]$.

[3] Remember that $R_j$ is the relation $R_{b_j} \circ R^{sf}_{S_j}$ associated with $b_j; S_j$.

Hence, together with $R_j \subseteq R_j^+$, which follows from Fact 5.2.5, we obtain that $\mathfrak{M} \models \exists\xi''[\tau^{\lambda}(false)(\xi'') \wedge R_j^+(\xi'', \xi')]$; i.e., $\mathfrak{M} \models (\tau^{\lambda}(false) \circ R_j^+)(\xi')]$. Together with (i) and (iii), $\mathfrak{M} \models \pi(\lambda, \langle j \rangle)(\xi')$ follows and hence $\mathfrak{M} \models \exists v \prec w \cdot \pi(v)(\xi')$.

(2) $1 \leqslant \mathrm{lh}(s) < n$, so $s = \langle i_1, ..., i_k \rangle$ for some $i_1, ..., i_k$ with $\{i_1, ..., i_k\} \subseteq \{1, ..., n\}$ and $1 \leqslant k < n$. From $\mathfrak{M} \models \pi(w)(\xi)$ we derive $\mathfrak{M} \models (\tau^{\lambda}(false) \circ \mathrm{Usegment}(R_{i_1}, ..., R_{i_k}) \circ (\bigcup_{t=1}^{k} R_{i_t})^*)(\xi)$. Since $\mathrm{Usegment}(R_{i_1}, ..., R_{i_k}) \circ (\bigcup_{t=1}^{k} R_{i_t})^* \circ R_j = \mathrm{Usegment}(R_{i_1}, ..., R_{i_k}, R_j) \subseteq$ (Definition 6.1.1 and $j \neq i_1, ..., i_k$ for $j \in D_w$) $\subseteq \mathrm{Usegment}(R_{i_1}, ..., R_{i_k}, R_j) \circ (\bigcup_{t=1}^{k} R_{i_t} \cup R_j)^*$ (Fact 5.2.5), together with the fact that $\mathfrak{M} \models R_j(\xi, \xi')$ holds, it follows that $\mathfrak{M} \models \tau^{\lambda}(false) \circ \mathrm{Usegment}(R_{i_1}, ..., R_{i_k}, R_j)(\xi')$ holds, too. It follows together with (i) and (iii) that $\mathfrak{M} \models \pi(\lambda, \langle i_1, ..., i_k, j \rangle)(\xi')$ holds. Again, $\mathfrak{M} \models \exists v \prec w \cdot \pi(v)(\xi')$ follows.

(3) $\mathrm{lh}(s) = n$. From $\mathfrak{M} \models \pi(\lambda, s)(\xi)$ and Definition 7.1.3, the existence of a $\beta < \lambda$ such that $\mathfrak{M} \models \pi(\beta, \langle\ \rangle)(\xi)$ follows. As in case (1), $\mathfrak{M} \models \exists v \prec (\beta, \langle\ \rangle) \cdot \pi(v)(\xi')$, and so $\mathfrak{M} \models \exists v \prec (\lambda, \langle\ \rangle) \cdot \pi(v)(\xi')$. ∎

LEMMA 7.2.2 (Corresponding to premise (b) of Orna's rule). *Let $w \in W_{\varepsilon, n}$, $j \in St_w$; i.e., $R_j$ is a steady move. Suppose that $\mathfrak{M} \models r \supset (\neg\mathrm{SFAIR}(R_1, ..., R_n) \wedge ((\bigcup_{i=1}^{n} R_i)^* \circ \bigwedge_{i=1}^{n} \neg b_i) \to q)$ holds. Then $\mathfrak{M} \models [\pi(w) \wedge w \succ \bar{0} \wedge b_j]\ S_j [\exists v \preccurlyeq w \cdot \pi(v)]$ holds, too.*

*Proof.* We have to show that for all states $\xi, \xi'$ such that $\mathfrak{M} \models R_j(\xi, \xi')$, $\mathfrak{M} \models (\pi(w) \wedge w \succ \bar{0})(\xi) \Rightarrow \mathfrak{M} \models \exists v \preccurlyeq w \cdot \pi(v)(\xi')$. To do so choose states $\xi, \xi'$ and suppose that $\mathfrak{M} \models (\pi(w) \wedge w \succ \bar{0})(\xi)$ holds. Let $w = (\lambda, s)$. As in Lemma 7.2.1 there are two cases:

(a) $\mathfrak{M} \models \bigwedge_{i=1}^{n} \neg b_i(\xi')$. In this case the lemma is trivial.

(b) $$\mathfrak{M} \models \bigvee_{i=1}^{n} b_i(\xi'). \tag{i}$$

We have to prove that $\mathfrak{M} \models \exists v \preccurlyeq w \cdot \pi(v)(\xi')$ is satisfied. Note that $\mathrm{lh}(s) \neq 0$ and $\mathrm{lh}(s) \neq n$, because $\mathrm{lh}(s) = 0$ or $\mathrm{lh}(s) = n$ implies that $St_w = \emptyset$. So let $w = (\lambda, \langle i_1, ..., i_k \rangle)$, $1 \leqslant k < n$, $\{i_1, ..., i_k\} \subseteq \{1, ..., n\}$. Since $j \in St_w$, $j = i_t$ for some $t$, $1 \leqslant t \leqslant k$. Now, $\mathfrak{M} \models \pi(w)(\xi)$, so $\mathfrak{M} \models \tau^{\lambda}(false) \circ \mathrm{Usegment}(R_{i_1}, ..., R_{i_k}) \circ (\bigcup_{t=1}^{k} R_{i_t})^*(\xi)$; i.e.,

$$\mathfrak{M} \models \exists\xi'' \cdot \left[\tau^{\lambda}(false)(\xi'') \wedge \mathrm{Usegment}(R_{i_1}, ..., R_{i_k}) \circ \left(\bigcup_{t=1}^{k} R_{i_t}\right)^* (\xi'', \xi)\right]. \tag{ii}$$

Since $(\bigcup_{l=1}^{k} R_{i_l})^* \circ R_j \subseteq (\bigcup_{l=1}^{k} R_{i_l})^*$, see Fact 5.2.5, we obtain that $\text{Usegment}(R_{i_1}, ..., R_{i_l}) \circ (\bigcup_{l=1}^{k} R_{i_l})^* \circ R_j \subseteq \text{Usegment}(R_{i_1}, ..., R_{i_l}) \circ (\bigcup_{l=1}^{k} R_{i_l})^*$. From (ii) and the fact that $\mathfrak{M} \models R_j(\xi, \xi')$, it follows that $\mathfrak{M} \models \exists\xi'[\tau^{\lambda}(\mathit{false})(\xi'') \wedge \text{Usegment}(R_{i_1}, ..., R_{i_k}) \circ (\bigcup_{l=1}^{k} R_{i_l})^* (\xi'', \xi')]$; i.e.,

$$\mathfrak{M} \models \left[\left(\tau^{\lambda}(\mathit{false}) \circ \text{Usegment}(R_{i_1}, ..., R_{i_k}) \circ \left(\bigcup_{l=1}^{k} R_{i_l}\right)^*\right)(\xi')\right]. \qquad \text{(iii)}$$

Moreover, as in the proof of Lemma 7.2.1, we see that

$$\mathfrak{M} \models r \circ \left(\bigcup_{i=1}^{n} R_i\right)^* (\xi') \qquad \text{holds, too.} \qquad \text{(iv)}$$

Now, (i), (iii), and (iv) imply $\mathfrak{M} \models \pi(\lambda, \langle i_1, ..., i_k \rangle)(\xi')$, whence $\mathfrak{M} \models \exists v \preccurlyeq w \cdot \pi(v)(\xi')$. ∎

The following lemma shows that clause (c) of Orna's rule is satisfied, too, under the assumption that $[r]\ {}^*[\Box_{i=1}^{n} b_i \rightarrow S_i][q]_{sf}$ holds.

LEMMA 7.2.3 (Corresponding to premise (c) of Orna's rule). *Suppose that* $\mathfrak{M} \models r \supset (\neg \text{SFAIR}(R_1, ..., R_n) \wedge (((\bigcup_{i=1}^{n} R_i)^* \circ \bigwedge_{i=1}^{n} \neg b_i) \rightarrow q))$ *holds. Then* $\mathfrak{M} \models [\pi(w) \wedge w \succ \bar{0}]\ {}^*[\Box_{i \in St_w} b_i \wedge \bigwedge_{j \in D_w} \neg b_j \rightarrow S_i][\mathit{true}]$ *holds, too.*

*Proof.* Observe that for all $w \in W_{\alpha,n}$ such that $w \succ \bar{0}$, $D_w \neq \emptyset$. So $St_w \subsetneq \{1, ..., n\}$. It follows that the program $S' \equiv {}^*[\Box_{i \in St_w} b_i \wedge \bigwedge_{j \in D_w} \neg b_j \rightarrow S_i]$ contains less directions than the original program. Therefore, we may apply the induction hypothesis. If $St_w = \emptyset$ then by convention $S' \equiv \mathbf{skip}$, in which case the lemma is trivial. So assume $St_w \neq \emptyset$.

After a possible renumbering, we may assume, too, that $St_w = \{1, ..., k\}$, $1 \leqslant k < n$. So, $D_w = \{k+1, ..., n\}$. Let $b'$ denote $\bigwedge_{j \in D_w} \neg b_j = \bigwedge_{j=k+1}^{n} \neg b_j$, and let $R'_i = b' \circ R_i$. By Theorem 6.3.4, and Corollary 6.3.5 we obtain that $\mathfrak{M} \models (\pi(w) \wedge w \succ \bar{0}) \supset \neg\text{SFAIR}(R'_1, ..., R'_k)$ implies $\mathfrak{M} \models [\pi(w) \wedge w \succ \bar{0}]\ {}^*[\Box_{i=1}^{k} b_i \wedge \bigwedge_{j=k+1}^{n} \neg b_j \rightarrow S_i][\mathit{true}]$ holds.

So, to prove the lemma, it suffices to show that $\mathfrak{M} \models (\pi(w) \wedge w \succ \bar{0}) \supset \neg\text{SFAIR}(R'_1, ..., R'_k)$. This follows from the next two claims.

CLAIM 1. *Under the aforementioned assumptions,* $\mathfrak{M} \models (\pi(w) \wedge w \succ \bar{0}) \supset \neg\text{UF}(R'_1, ..., R'_k)$ *holds.*

*Proof of Claim* 1. Suppose that $\mathfrak{M} \models \pi(w)(\xi) \wedge w \succ \bar{0}$ holds. Then $\mathfrak{M} \models r \circ (\bigcup_{i=1}^{n} R_i)^* (\xi)$, i.e., $\mathfrak{M} \models \exists\xi'' \cdot [r(\xi'') \wedge (\bigcup_{i=1}^{n} R_i)^* (\xi'', \xi)]$ holds, too. As a consequence of our assumptions, we obtain that $\mathfrak{M} \models r \supset \neg\text{SFAIR}(R_1, ..., R_n)$ and so $\mathfrak{M} \models \exists\xi'' \cdot [\neg\text{SFAIR}(R_1, ..., R_n)(\xi'') \wedge (\bigcup_{i=1}^{n} R_i)^* (\xi'', \xi)]$. Thus, $\mathfrak{M} \models \exists\xi'' \cdot [((\bigcup_{i=1}^{n} R_i)^* \rightarrow \neg\text{UF}((\bigwedge_{i=k+1}^{n} \neg b_i)$

$\circ R_1, \ldots, (\bigwedge_{i=k+1}^{n} \neg b_i) \circ R_k))(\xi'') \wedge (\bigcup_{i=1}^{n} R_i)^* (\xi'', \xi)]$ holds by Lemma 6.2.3. Consequently, $\mathfrak{M} \models \exists \xi'' \cdot [((\bigcup_{i=1}^{n} R_i)^* \to \neg \mathrm{UF}(R'_1, \ldots, R'_k))(\xi'') \wedge (\bigcup_{i=1}^{n} R_i)^* (\xi'', \xi)]$, from which $\mathfrak{M} \models \neg \mathrm{UF}(R'_1, \ldots, R'_k)(\xi)$ follows by definition of $R \to p$. This proves Claim 1.

Now, if $k = 1$, the lemma follows immediately from Claim 1 and Definition 6.2.2. So assume that $k \geqslant 2$.

CLAIM 2. *Under the aforementioned assumptions,*

$$\mathfrak{M} \models (\pi(w) \wedge w \succ \bar{0}) \supset \bigwedge_{\substack{i_1, \ldots, i_k \text{ perm of } 1, \ldots, k \\ 1 \leqslant l < k}} \neg \mathrm{fair}(R'_{i_1}, \ldots, R'_{i_l})$$

$$\mathrm{fin}(R'_{i_{l+1}}, \ldots, R'_{i_k}) \text{ holds.}$$

*Proof of Claim* 2. Let $1 \leqslant l < k$. For simplicity, we shall prove that $\mathfrak{M} \models (\pi(w) \wedge w \succ \bar{0}) \supset \neg \mathrm{fair}(R'_1, \ldots, R'_l) \, \mathrm{fin}(R'_{l+1}, \ldots, R'_k)$, since any other permutation is treated in a similar way. By Definition 6.2.1, we must show that

$$\mathfrak{M} \models (\pi(w) \wedge w \succ \bar{0}) \supset \left( \left( \bigcup_{i=1}^{k} R'_i \right)^* \to \neg \mathrm{UF} \left( \left( \neg b' \vee \bigwedge_{i=l+1}^{k} \neg b_i \right) \right.\right.$$

$$\left.\left. \circ R'_1, \ldots, \left( \neg b' \vee \bigwedge_{i=l+1}^{k} \neg b_i \right) \circ R'_l \right) \right)$$

holds. This is a consequence of the following chain of implications:

$$\mathfrak{M} \models (\pi(w) \wedge w \succ \bar{0})(\xi)$$

$$\Rightarrow \mathfrak{M} \models r \circ \left( \bigcup_{i=1}^{n} R_i \right)^* (\xi) \qquad \text{(Definition 7.1.4)}$$

$$\Rightarrow \mathfrak{M} \models \exists \xi'' \cdot \left[ r(\xi'') \wedge \left( \bigcup_{i=1}^{n} R_i \right)^* (\xi'', \xi) \right]$$

(Definition 5.5.5)

$$\Rightarrow \mathfrak{M} \models \exists \xi'' \cdot \left[ \neg \mathrm{SFAIR}(R_1, \ldots, R_n)(\xi'') \wedge \left( \bigcup_{i=1}^{n} R_i \right)^* (\xi'', \xi) \right]$$

(by assumptions)

$$\Rightarrow \mathfrak{M} \models \exists \xi'' \cdot \left[ \left( \left( \bigcup_{i=1}^{n} R_i \right)^* \to \neg \mathrm{UF} \left( \bigwedge_{j=l+1}^{n} \neg b_j \circ R_1, \ldots, \right.\right.\right.$$

$$\left.\left.\left. \bigwedge_{j=l+1}^{n} \neg b_j \circ R_l \right) \right) (\xi'') \wedge \left( \bigcup_{i=1}^{n} R_i \right)^* (\xi'', \xi) \right]. \quad (*)$$

The latter implication follows from Lemma 6.2.3. Hence, for all $t = 1, ..., l$,

$$\begin{aligned}
&\left(\neg b' \vee \bigwedge_{i=l+1}^{k} \neg b_i\right) \circ R_t' \\
&\quad = \left(\neg b' \vee \bigwedge_{i=l+1}^{k} \neg b_i\right) \wedge b' \circ R_t \\
&\quad = \left(b' \wedge \bigwedge_{i=l+1}^{k} \neg b_i\right) \circ R_t \\
&\quad = \left(\bigwedge_{i=k+1}^{n} \neg b_i \wedge \bigwedge_{i=l+1}^{k} \neg b_i\right) \circ R_t \\
&\quad = \bigwedge_{i=l+1}^{n} \neg b_i \circ R_t \qquad (\text{since } l+1 \leqslant k < n).
\end{aligned}$$

So, $(*)$ implies that $\mathfrak{M} \models \exists \xi'' \cdot [((\bigcup_{i=1}^{n} R_i)^* \to \neg \mathrm{UF}((\neg b' \vee \bigwedge_{i=l+1}^{k} \neg b_i) \circ R_1', ..., (\neg b' \vee \bigwedge_{i=l+1}^{k} \neg b_i) \circ R_l'))(\xi'') \wedge (\bigcup_{i=1}^{n} R_i)^* (\xi'', \xi)]$, and finally $\mathfrak{M} \models ((\bigcup_{i=1}^{n} R_i')^* \to \neg \mathrm{UF}((\neg b' \vee \bigwedge_{i=l+1}^{k} \neg b_i) \circ R_1', ..., (\neg b' \vee \bigwedge_{i=l+1}^{k} \neg b_i) \circ R_l'))(\xi)$ by using Fact 5.2.5. As an immediate consequence, we then obtain that $\mathfrak{M} \models ((\bigcup_{i=1}^{k} R_i')^* \to \neg \mathrm{UF}((\neg b' \vee \bigwedge_{i=l+1}^{k} \neg b_i) \circ R_1', ..., (\neg b' \vee \bigwedge_{i=l+1}^{k} \neg b_i) \circ R_l'))(\xi)$ holds, too. This proves Claim 2 and hence the lemma. ∎

It remains to show that clause (d) of Orna's rule is satisfied, too. This is established in the following

LEMMA 7.2.4 (Corresponding to clause (d) of Orna's rule). *Suppose that* $\mathfrak{M} \models r \supset (\neg \mathrm{SFAIR}(R_1, ..., R_n) \wedge ((\bigcup_{i=1}^{n} R_i)^* \circ \bigwedge_{i=1}^{n} \neg b_i) \to q)$ *holds. Then* (a), (b), *and* (c) *below hold, too.*

(a) $\mathfrak{M} \models r \supset (\exists v \cdot \pi(v))$.

(b) $\mathfrak{M} \models (\pi(w) \wedge w \succ \bar{0}) \supset \bigvee_{i=1}^{n} b_i$.

(c) $\mathfrak{M} \models \pi(\bar{0}) \supset ((\bigwedge_{i=1}^{n} \neg b_i) \wedge q)$.

*Proof.* (a) Let $\xi \in \mathrm{States}$ satisfy $\mathfrak{M} \models r(\xi)$. If $\mathfrak{M} \models \bigwedge_{i=1}^{n} \neg b_i(\xi)$, then we are done, because $\mathfrak{M} \models \pi(\bar{0})(\xi)$ holds. Hence, let

$$\mathfrak{M} \models \bigvee_{i=1}^{n} b_i(\xi). \tag{i}$$

Clearly,

$$\mathfrak{M} \models r \circ \left(\bigcup_{i=1}^{n} R_i\right)^* (\xi) \qquad \text{holds.} \tag{ii}$$

Since $\mathfrak{M} \models r(\xi)$ holds, $\mathfrak{M} \models \neg \mathrm{SFAIR}(R_1, ..., R_n)(\xi)$ holds, too, and consequently,

$$\mathfrak{M} \models \neg \mathrm{UF}(R_1, ..., R_n)(\xi); \qquad \text{i.e.,} \quad \mathfrak{M} \models \tau^{\sigma}(\mathit{false})(\xi). \qquad \text{(iii)}$$

It follows from (i), (ii), (iii) that $\mathfrak{M} \models \pi(\bar{\alpha}, \langle\ \rangle)(\xi)$ holds.

(b) This immediately follows from Definition 7.1.4.

(c) From Definition 7.1.4 it follows that $\mathfrak{M} \models \pi(\bar{0}) \supset \bigwedge_{i=1}^{n} \neg b_i$. Therefore, it remains to show that $\mathfrak{M} \models \pi(\bar{0}) \supset q$. To do so, choose some $\xi$ with $\mathfrak{M} \models \pi(\bar{0})(\xi)$. By Definition 7.1.4, there exists some $\xi'$ satisfying $\mathfrak{M} \models r(\xi') \wedge (\bigcup_{i=1}^{n} R_i)^* (\xi', \xi)$. Since $\mathfrak{M} \models r \supset (((\bigcup_{i=1}^{n} R_i)^* \circ \bigwedge_{i=1}^{n} \neg b_i) \rightarrow q)$ holds by assumption, the implication to be proved now immediately follows. ∎

THEOREM 7.2.5 (Completeness of our proof system). *For all assertions $r$, $q$, commands $S$, $\mathfrak{M} \models [r]\ S[q]_{sf}$ implies* $\mathrm{Th}(\mathfrak{M}) \vdash [r]\ S[q]$.

*Proof.* Clearly, the only non-trivial case is when $S \equiv {}^*[\Box_{i=1}^{n} b_i \rightarrow S_i]$ for $n \geqslant 2$. We have to show that for all assertions $r, q$, $\mathfrak{M} \models [r]\ {}^*[\Box_{i=1}^{n} b_i \rightarrow S_i][q]_{sf} \Rightarrow \mathrm{Th}(\mathfrak{M}) \vdash [r]\ {}^*[\Box_{i=1}^{n} b_i \rightarrow S_i][q]$ holds. This is, however, an immediate consequence of the induction hypothesis, Theorem 6.3.4, Section 4, Definitions 7.1.3 through 7.1.5, and the Lemmata 7.2.1 through 7.2.4. ∎

## 8. SOUNDNESS

In this section we prove the soundness of our proof system, i.e., for all assertions $r$, $q$ and command $S$, $\mathrm{Th}(\mathfrak{M}) \vdash [r]\ S[q] \Rightarrow \mathfrak{M} \models [r]\ S[q]_{sf}$.

It is obvious that the rules for assignment, consequence, and sequential composition are sound. Therefore it remains to prove the soundness of Orna's rule. Let $S \equiv {}^*[\Box_{i=1}^{n} b_i \rightarrow S_i]$. In case $n = 1$ Orna's rule reduces to Harel's rule for terminating loops proved sound in Harel (1979). Consequently, assume that $n \geqslant 2$ holds. We may assume, too, that the following induction hypothesis (IH) holds:

—For all simple commands $S$, $\mathrm{Th}(\mathfrak{M}) \vdash [r]\ S[q] \Rightarrow \mathfrak{M} \models [r]\ S[q]_{sf}$, and

—For all $k$ with $1 \leqslant k < n$, $\mathrm{Th}(\mathfrak{M}) \vdash [r]\ {}^*[\Box_{i=1}^{k} b_i \rightarrow S_i][q] \Rightarrow \mathfrak{M} \models [r]\ {}^*[\Box_{i=1}^{k} b_i \rightarrow S_i][q]_{sf}$.

Next assume $\mathrm{Th}(\mathfrak{M}) \vdash [r]\ S[q]$. We have to prove that $\mathfrak{M} \models [r]\ S[q]_{sf}$

holds. To do so, it suffices to show that $\mathfrak{M} \models r \supset \text{wpsf}(S, q)$ which by Definition 6.3.3 and Theorem 6.3.4 amounts to proving

$$\mathfrak{M} \models r \supset \left(\neg \text{SFAIR}(R_1, ..., R_n) \wedge \left(\left(\left(\bigcup_{i=1}^{n} R_i\right)^* \circ \bigwedge_{i=1}^{n} \neg b_i\right) \to q\right)\right).$$

By Lemma 6.2.3 $\mathfrak{M} \models \neg \text{SFAIR}(R_1, ..., R_n) \Leftrightarrow \mathfrak{M} \models (\neg \text{UF}(R_1, ..., R_n) \wedge \bigwedge_{i_1, ..., i_n \text{ perm of } 1, ..., n;\ 1 \leq k < n} (\bigcup_{i=1}^{n} R_i)^* \to \neg \text{UF}(\bigwedge_{j=k+1}^{n} \neg b_{i_j} \circ R_{i_1}, ..., \bigwedge_{j=k+1}^{n} \neg b_{i_j} \circ R_{i_k}))$. Consequently, we have to show that $\mathfrak{M} \models r \supset \neg \text{UF}(R_1, ..., R_n)$,

$$\mathfrak{M} \models r \supset \bigwedge_{\substack{i_1, ..., i_n \text{ perm of } 1, ..., n \\ 1 \leq k < n}} \left(\bigcup_{i=1}^{n} R_i\right)^*$$
$$\to \neg \text{UF}\left(\bigwedge_{j=k+1}^{n} \neg b_{i_j} \circ R_{i_1}, ..., \bigwedge_{j=k+1}^{n} \neg b_{i_j} \circ R_{i_k}\right),$$

and $\mathfrak{M} \models r \supset (((\bigcup_{i=1}^{n} R_i)^* \circ \bigwedge_{i=1}^{n} \neg b_i) \to q)$ hold. These are established in Theorems 8.1, 8.2, and 8.3 below.

LEMMA 8.1. *Assume that* $\text{Th}(\mathfrak{M}) \vdash [r]\ {*}[\square_{i=1}^{n} b_i \to S_i][q]$ *holds. Then* $\mathfrak{M} \models r \supset \neg \text{UF}(R_1, ..., R_n)$ *holds, too.*

*Proof.* Let $\mathfrak{M} \models r(\xi)$ and suppose, to obtain a contradiction that, $\mathfrak{M} \models \text{UF}(R_1, ..., R_n)(\xi)$ holds. Since $D_w \neq \emptyset$ for $w > \bar{0}$, there exists an infinite decreasing sequence in $W$, starting in some $w \in W$ such that $\mathfrak{M} \models \pi(w)(\xi)$ holds. This contradicts the well-foundedness of $W$. ∎

Next, as a preparation for Lemma 8.2 we first prove the following claim that captures the most difficult part of that lemma.

CLAIM. *Assume that* $\text{Th}(\mathfrak{M}) \vdash [r]\ {*}[\square_{i=1}^{n} b_i \to S_i][q]$ *holds. Let* $\xi$ *be a state such that* $\mathfrak{M} \models r(\xi)$ *holds. For all* $\xi'$ *satisfying* $\mathfrak{M} \models (\bigcup_{i=1}^{n} R_i)^* (\xi, \xi')$, $\mathfrak{M} \models \neg \text{UF}(b' \circ R_1, ..., b' \circ R_k)(\xi')$ *holds, where* $b' = \bigwedge_{i=k+1}^{n} \neg b_i$.

*Proof.* Assume that the claim is false; i.e., there exist states $\xi$ and $\xi'$ such that $\mathfrak{M} \models (\bigcup_{i=1}^{n} R_i)^* (\xi, \xi')$ and $\mathfrak{M} \models \text{UF}(b' \circ R_1, ..., b' \circ R_k)(\xi')$ hold. Both $\xi$ and $\xi'$ are accessible states; i.e., both $\mathfrak{M} \models r \circ (\bigcup_{i=1}^{n} R_i)^* (\xi)$ and $\mathfrak{M} \models r \circ (\bigcup_{i=1}^{n} R_i)^* (\xi')$ hold. From the assumption that $\mathfrak{M} \models \text{UF}(b' \circ R_1, ..., b' \circ R_k)(\xi')$ holds, we infer the existence of an infinite strongly fair sequence of moves $b' \circ R_1, ..., b' \circ R_k$. As a consequence of the assumption that $\text{Th}(\mathfrak{M}) \vdash [r]\ {*}[\square_{i=1}^{n} b_i \to S_i][q]$ holds, we conclude that Orna's rule has been applied. Consequently, related to the infinite strongly fair sequence of moves, whose existence we showed above, is an infinite sequence $w_1, w_2, w_3, \ldots$ in $W$ such that $\mathfrak{M} \models \pi(w_1)(\xi')$ and for all $i \geqslant 0$ $w_i \geqslant w_{i+1}$ hold. Since $W$ is well founded we obtain that there exists some $j \geqslant 0$ such that for all $i \geqslant j$ $w_i = w_{i+1}$. This implies that eventu... none of the moves taken in the infinite strongly fair sequence are decreasing moves. Furthermore, there exists a state $\xi''$ such that

(a) $\mathfrak{M} \models (\text{Upart}(b' \circ R_1, \ldots, b' \circ R_k))^* (\xi', \xi'')$,

(b) $\mathfrak{M} \models \text{UF}(b' \circ R_1, \ldots, b' \circ R_k)(\xi'')$, and

(c) there exist a $w''$, $w''$ not minimal, satisfying $w'' \leqslant w_1$, $\mathfrak{M} \models \pi(w'')(\xi'')$, and $\{1, \ldots, k\} \subseteq St_{w''}$.

Let $St_{w''} = \{j_1, \ldots, j_{k+m}\}$ for some $m \geqslant 0$, where $j_t = t$, for $t = 1, \ldots, k$. Note that this implies that $D_{w''} = \{j_{k+m+1}, \ldots, j_n\} = \{1, \ldots, n\} - St_{w''}$ holds. Now, $w'' > 0$ and $\text{Th}(\mathfrak{M}) \vdash [\pi(w'') \wedge w'' > 0]^* [\Box_{i \in St_{w''}} b_i \wedge \bigwedge_{j \in D_{w''}} \neg b_j \rightarrow S_i]$ $[true]$ holds by the third clause of Orna's rule. Hence, as a consequence of the induction hypothesis and the fact that $\mathfrak{M} \models (\pi(w'') \wedge w'' > 0)(\xi'')$, we obtain that

$$\mathfrak{M} \models \neg\text{SFAIR}\left( \bigwedge_{t=k+m+1}^{n} \neg b_{j_t} \circ R_{j_1}, \ldots, \bigwedge_{t=k+m+1}^{n} \neg b_{j_t} \circ R_{j_{k+m}} \right)(\xi''), \quad \text{(i)}$$

i.e., there does not exist an infinite strongly fair sequence of steady moves in which no decreasing move is ever enabled. To obtain a contradiction, we now distinguish two cases:

(A) $m = 0$. Then (i) implies that $\mathfrak{M} \models \neg\text{UF}(b' \circ R_1, \ldots, b' \circ R_k)(\xi'')$ as $j_t = t$ for $1 \leqslant t \leqslant k$. This follows from Definition 6.2.3, and contradicts (b).

(B) $m \neq 0$. Note that for all $s$, $k+1 \leqslant s \leqslant k+m$, the actual enablings-condition for $\bigwedge_{t=k+m+1}^{n} \neg b_{j_t} \circ R_s$ is $\bigwedge_{t=k+m+1}^{n} \neg b_{j_t} \wedge b_s$. By (i) and Definition 6.2.2 $\mathfrak{M} \models \neg\text{fair}(\bigwedge_{t=k+m+1}^{n} \neg b_{j_t} \circ R_1, \ldots, \bigwedge_{t=k+m+1}^{n} \neg b_{j_t} \circ R_k)$ $\text{fin}(\bigwedge_{t=k+m+1}^{n} \neg b_{j_t} \circ R_{j_{k+1}}, \ldots, \bigwedge_{t=k+m+1}^{n} \neg b_{j_t} \circ R_{j_{k+m}})(\xi'')$ holds. So by Definition 6.2.1, $\mathfrak{M} \models (\bigcup_{l=1}^{k+m} (\bigwedge_{t=k+m+1}^{n} \neg b_{j_t}) \circ R_{j_l})^* \rightarrow \neg\text{UF}(C \circ R_1, \ldots, C \circ R_k)$ $(\xi'')$ holds, too, where $C = \bigwedge_{t=k+m+1}^{n} \neg b_{j_t} \wedge \bigwedge_{s=k+1}^{k+m} \neg(\bigwedge_{t=k+m+1}^{n} \neg b_{j_t} \wedge b_{j_s})$. Hence, we obtain $\mathfrak{M} \models \neg\text{UF}(C \circ R_1, \ldots, C \circ R_k)(\xi'')$. As $\mathfrak{M} \models C = \bigwedge_{t=k+1}^{n} \neg b_t$, this implies $\mathfrak{M} \models \neg\text{UF}(\bigwedge_{t=k+1}^{n} \neg b_j \circ R_1, \ldots, \bigwedge_{t=k+1}^{n} \neg b_j \circ R_k)(\xi'')$, again contradicting (b).

This proves the claim. ▮

LEMMA 8.2. *Assume that* $\text{Th}(\mathfrak{M}) \vdash [r]^* [\Box_{i=1}^{n} b_i \rightarrow S_i][q]$ *holds. Let* $k$ *be given,* $1 \leqslant k < n$*, and assume furthermore that* $i_1, \ldots, i_n$ *is some permutation of* $1, \ldots, n$*. Then* $\mathfrak{M} \models r \supset \neg\text{fair}(R_{i_1}, \ldots, R_{i_k})\,\text{fin}(R_{i_{k+1}}, \ldots, R_{i_n})$ *holds, too.*

*Proof.* Possibly, after a renumbering, let $i_1, \ldots, i_n$ be the identity permutation of $1, \ldots, n$. Hence, we show that $\mathfrak{M} \models \neg\text{fair}(R_1, \ldots, R_k)$

$\mathrm{fin}(R_{k+1}, \ldots, R_n)(\xi)$ holds, where $\xi$ satisfies $r$. According to Definition 6.2.1, it suffices to prove that for all $\xi'$ satisfying $\mathfrak{M} \models (\bigcup_{i=1}^{n} R_i)^* (\xi, \xi')$, $\mathfrak{M} \models \neg \mathrm{UF}(b' \circ R_1, \ldots, b' \circ R_k)(\xi')$ holds, where $b' = \bigwedge_{i=k+1}^{n} \neg b_i$. This immediately follows from the claim above and establishes the theorem. ∎

LEMMA 8.3. *Assume that* $\mathrm{Th}(\mathfrak{M}) \vdash [r]\ {*}[\Box_{i=1}^{n} b_i \to S_i][q]$ *holds. Then* $\mathfrak{M} \models r \supset ((\bigcup_{i=1}^{n} R_i)^* \circ \bigwedge_{i=1}^{n} \neg b_i \to q)$ *holds, too.*

*Proof.* This lemma is trivial. ∎

Finally, we arrive at the main theorem of this section, stating the soundness of our proof system. Its proof is straightforward now.

THEOREM 8.3 (Soundness of the proof system). *For all assertions* $r, q$, *commands* $S$, $\mathrm{Th}(\mathfrak{M}) \vdash [r]\ S[q] \Rightarrow \mathfrak{M} \models [r]\ S[q]_{sf}$ *holds.*

*Proof.* The only non-trivial case is when $S \equiv {*}[\Box_{i=1}^{n} b_i \to S_i]$ and $n \geqslant 2$. Consequently, we have to prove that $\mathrm{Th}(\mathfrak{M}) \vdash [r]\ {*}[\Box_{i=1}^{n} b_i \to S_i][q] \Rightarrow \mathfrak{M} \models [r]\ {*}[\Box_{i=1}^{n} b_i \to S_i][q]_{sf}$ holds for $n \geqslant 2$. This follows from Lemmata 8.1, 8.2, 8.3, (IH), Definition 6.3.3, and Theorem 6.3.4. ∎

## 9. How to Deal with Nested Repetitions

In the previous sections we have considered a rather simple programming language. E.g., according to the syntax given in Section 2 repetitions never contained inner repetitions. In this section we shall drop this restriction and outline how to deal with the more general case. Basically, we proceed as before, adjusting the definitions and theorems to deal with a less restrictive programming language.

### 9.1. *Basic Ideas*

Until now, we have considered a very simple programming language, in which, in repetition $S \equiv {*}[\Box_{i=1}^{n} b_i \to S_i]$, $n \geqslant 1$, the $S_i$ consisted of finite sequences of assignments ($i = 1, \ldots, n$). According to the syntax given in Section 2, the program

$$
\begin{array}{l}
S_2 \equiv {*}[b_1 \to {*}[b_2 \to x := x+1 \\
\qquad\qquad\quad \Box b_2 \to b_2 := \mathit{false} \\
\qquad\qquad\quad ] \\
\quad\ \Box b_1 \to b_1 := \mathit{false} \\
\quad\ ]
\end{array}
$$

is not allowed. The reason for disallowing nested repetitions is the possibility of (strongly fair) divergence of inner loops, which slightly complicates the earlier theorems.

Intuitively speaking, the program $S_2$ above should terminate strongly fair, when this notion is suitably refined: if execution of $S_2$ starts in a state satisfying both $b_1$ and $b_2$—the other cases are trivial and omitted—$S_2$ terminates as soon as direction 2, i.e., $b_1; b_1 := \mathit{false}$, is taken. Under the strong fairness, as defined below, this direction must be chosen eventually because the inner loop $*[b_2 \to x := x+1 \;\Box b_2 \to b_2 := \mathit{false}]$ terminates strongly fair. To gain a better understanding of this notion, consider the program below. It does not terminate strongly fair according to the definition of strongly fair termination (see Definition 9.4 below).

```
*[b₁ → b₂ := true
□b₁ → *[b₂ → b₁ := false
        □b₂ → b₂ := false
        ]
].
```

Starting in a state in which $b_1$ holds, executing the first direction, i.e., $b_1; b_2 := \mathit{true}$, followed by executing the second direction, in which in the inner loop the second direction always is chosen, i.e., $b_1; (b_2; b_2 := \mathit{false})$, constitutes a strongly fair computation (according to the definition below). Each of the loops is treated strongly fair *whenever entered*. However, strong fairness does not constrain choices that are made in consecutive executions of the same loop. This program would terminate under yet another fairness assumption; viz., that of all-level (global) fairness (Apt *et al.*, 1984).

In this section we briefly outline how to deal with a less restrictive language, LGC′($\mathfrak{M}$) in which nested repetitions are allowed. Again, we assume a given signature and a first-order structure $\mathfrak{M}$ as above. The syntax of the less restricted language is given by the following BNF-productions:

$\langle$command$\rangle$ ::= $\langle$assignment$\rangle$ | $\langle$composition$\rangle$ | $\langle$repetition$\rangle$.

$\langle$assignment$\rangle$ ::= $\langle$variable$\rangle$ := $\langle$expression$\rangle$.

$\langle$composition$\rangle$ ::= $\langle$command$\rangle$; $\langle$command$\rangle$.

$\langle$repetition$\rangle$ ::= *[{□selection}].

$\langle$selection$\rangle$ ::= $\langle$guard$\rangle$ → $\langle$command$\rangle$.

$\langle$guard$\rangle$ ::= "quantifier-free boolean expression."

Again, *[ ] is identified with **skip** and $*[\Box_{i=1}^{n} b_i \to S_i]$ abbreviates $*[\Box b_1 \to S_1 \cdots \Box b_n \to S_n]$ $(n \geq 1)$.

As before, four semantics, viz., $R_S^{\text{part}}$, $R_S^{t}$, $R_S^{\text{uf}}$, $R_S^{\text{sf}}$, for $S \in$ LGC′($\mathfrak{M}$) are defined. The case $R_S^{\text{part}}$ is essentially the same as in Section 3 and is therefore omitted. For the other cases the possibility of divergence within some branch will now have to be taken into account.

Let States denote the set of states and let $\perp$ denote the divergence state. In the sequel it is assumed that $\perp \in$ States and that for each relation $R \subseteq$ States$^2$, $\forall \xi \cdot [R(\perp, \xi) \Rightarrow \xi = \perp]$ holds. For assertions $p$, $p(\perp) = \mathit{false}$, i.e., $p$ never holds in $\perp$.

The definitions of the various semantics, as well as the soundness and completeness proofs will use induction on the level of statements:

DEFINITION 9.2 (Level of statements). The level of an assignment $x := e$ is 0. Let the levels of $S_i$ be $k_i$ $(i = 1, 2)$. Then $S_1 ; S_2$ has level $\max(k_1, k_2)$. Let $S \equiv *[\Box_{i=1}^{n} b_i \rightarrow S_i]$, with $n \geqslant 1$. Then the level of $S$ is $1 + \max\{k_i | 1 \leqslant i \leqslant n\}$, where $S_i$ has level $k_i$ for $i = 1, ..., n$.

DEFINITION 9.3 ($R'_S$). For $S \in \mathrm{LGC}'(\mathfrak{M})$, the relation $R'_S$ is defined as follows:

$$R'_S = R_S^{\text{part}} \cup \{(\perp, \perp)\}, \qquad \text{if} \quad S \equiv x := e.$$

$$R'_S = R'_{S_1} \circ R'_{S_2}, \qquad \text{if} \quad S \equiv S_1 ; S_2.$$

To define $R'_S$ for repetitions $S$, again the notion of an execution sequence of $S$ is needed. Its definition is similar to Definition 3.2.1 and therefore omitted. $S$ is said to diverge nondeterministically from $\xi$, if there exists an execution sequence of $S$ starting in $\xi$ that is either infinite, or finite and ends in $\perp$.

Finally, define for $S \equiv *[\Box_{i=1}^{n} b_i \rightarrow S_i]$ with $n \geqslant 1$,

$$R'_S = R_S^{\text{part}} \cup \{(\xi, \perp) | S \text{ can diverge nondeterministically from } \xi\} \cup \{(\perp, \perp)\}.$$

Note that an execution sequence of a loop $S$ ends in $\perp$ when an inner loop of $S$ is executed which diverges nondeterministically.

We now proceed with defining strongly fair execution sequences for repetitions $S \equiv *[\Box_{i=1}^{n} b_i \rightarrow S_i]$ with $n \geqslant 1$. As the example of $S_2$ above shows, strong-fairness does not consider the choices made at the top-level only, i.e., choices between the $b_i$ $(i = 1, ..., n)$, but also the choices made between the guards of inner loops of $S$.

DEFINITION 9.4 (Strongly fair termination).

(i) Let $\xi$ denote a state, $\xi \neq \perp$. An assignment always terminates strongly fair from $\xi$. $S_1 ; S_2$ terminates strongly fair from $\xi$, if $S_1$ terminates strongly fair from $\xi$ and $S_2$ terminates strongly fair for all possible output states produced by strongly fair computations of $S_1$.[4]

Now, let $S \equiv *[\square_{i=1}^{n} b_i \rightarrow S_i]$, with $n \geqslant 1$. An execution sequence of $S$ starting in $\xi$, is strongly fair, if either

(a) it is finite (say $\xi_0, \xi_1, ..., \xi_m$, where $\xi = \xi_0$) and either $\xi_m \neq \perp$, or $\xi_m = \perp$ and there exists an $S_i$ $(i = 1, ..., n)$ which strongly fair diverges from $\xi_{m-1}$, or

(b) it is infinite and every direction in $S$, which is infinitely often enabled along the sequence is chosen infinitely often. We say that $S$ terminates strongly fair from $\xi$ if it admits neither infinite strongly fair execution sequences nor finite ones ending in $\perp$ that start in $\xi$.

(ii) A program terminates strongly fair if it terminates strongly fair from $\xi$, for every $\xi \neq \perp$.

(iii) A program is said to diverge strongly fair if it admits a strongly fair computation, starting in $\xi$ that is either infinite, or finite and ends in $\perp$.

DEFINITION 9.5 (Unconditionally fair termination).

(i) Let $\xi$ denote a state, $\xi \neq \perp$. An assignment always terminates unconditionally fair from $\xi$. $S_1; S_2$ terminates unconditionally fair from $\xi$, if $S_1$ terminates unconditionally fair from $\xi$ and $S_2$ terminates unconditionally fair for all possible output states produced by unconditionally fair computations of $S_1$.[5]

Now, let $S \equiv *[\square_{i=1}^{n} b_i \rightarrow S_i]$, with $n \geqslant 1$. An execution sequence of $S$ starting in $\xi$, is unconditionally fair, if either

(a) it is finite (say $\xi_0, \xi_1, ..., \xi_m$, where $\xi = \xi_0$) and either $\xi_m \neq \perp$, or $\xi_m = \perp$ and there exists an $S_i$ $(i = 1, ..., n)$ which unconditionally fair diverges from $\xi_{m-1}$, or

(b) it is infinite and every direction in $S$ is chosen infinitely often. We say that $S$ terminates unconditionally fair from $\xi$ if it admits neither infinite unconditionally fair execution sequences nor finite ones ending in $\perp$, that start in $\xi$.

(ii) A program terminates unconditionally fair if it terminates unconditionally fair from $\xi$, for every $\xi \neq \perp$.

(iii) A program is said to diverge unconditionally fair if it admits an unconditionally fair computation, starting in $\xi$ that is either infinite, or finite and ends in $\perp$.

[4] Although, we have not defined what output states produced by strongly fair computations are, this notion should be clear.

[5] Although, we have not defined what output states produced by unconditionally fair computations are, this notion should be clear.

It can be shown that the relation between the fairness assumptions as formulated in Theorem 3.3.4 still holds.

DEFINITION 9.6 ($R_S^{sf}$, $R_S^{uf}$).

$$R_S^{uf} = R_S^{sf} = R_S^t \qquad \text{for} \quad S \equiv x := e,$$

$$R_S^{uf} = R_{S_1}^{uf} \circ R_{S_2}^{uf} \text{ and } R_S^{sf} = R_{S_1}^{sf} \circ R_{S_2}^{sf} \qquad \text{for} \quad S \equiv S_1 ; S_2.$$

For $S \equiv {}^*[\Box_{i=1}^n b_i \to S_i]$ with $n \geqslant 1$, we define

$$R_S^{uf} = R_S^{part} \cup \{(\xi, \bot) \mid S \text{ can diverge unconditionally fair from } \xi\}$$
$$\cup \{(\bot, \bot)\}.$$
$$R_S^{sf} = R_S^{part} \cup \{(\xi, \bot) \mid S \text{ can diverge strongly fair from } \xi\}$$
$$\cup \{(\bot, \bot)\}.$$

As before, we define the notions of nondeterministic, unconditionally fair (resp. strongly fair), termination of a program $S$ by $\forall \xi \neq \bot \cdot \neg R_S^t(\xi, \bot)$, $\forall \xi \neq \bot \cdot \neg R_S^{uf}(\xi, \bot)$ (resp. $\forall \xi \neq \bot \cdot \neg R_S^{sf}(\xi, \bot)$).

Again, this gives us four notions of validity, $\mathfrak{M} \models [p]\, S[q]_s$, for $s \in \{\text{part, t, uf, sf}\}$ which are the same as formulated in Definitions 3.2.2, 3.3.2, and 3.4.6.

The proof system is similar to the one in Section 4, except that in the composition rule and in Orna's rule the restriction to simple commands is dropped.

We now proceed to define a formula $F(R_S)$ such that for any state $\xi$, $F(R_S)(\xi)$ holds iff $S$ terminates strongly fair when execution of $S$ is started in $\xi$. Clearly, if $S$ is a loop, the formula $\neg$SFAIR does not suffice any more to describe the absence of infinite strongly fair execution sequences of $S$, since this formula only constrains choices made at the outermost level of the repetition. We now need a formula that also constrains the choices made in inner loops.

DEFINITION 9.7. The formula $F(R)$ is inductively defined as

$$F(R_S^{sf}) = \lambda\xi \cdot true, \qquad \text{if} \quad S \equiv x := e.$$

$$F(R_S^{sf}) = F(R_{S_1}^{sf}) \wedge (R_{S_1}^{sf} \to F(R_{S_2}^{sf})), \qquad \text{if} \quad S \equiv S_1 ; S_2.$$

Finally, if $S \equiv {}^*[\Box_{i=1}^n b_i \to S_i]$ $(n \geqslant 1)$, then $F(R_S^{sf}) = ((\bigcup_{i=1}^n R_i)^* \to \bigwedge_{i=1}^n (b_i \supset F(R_{S_i}^{sf}))) \wedge \neg\text{SFAIR}(R_1, ..., R_n)$; i.e., whenever the $i$th direction is taken along an execution sequence of $S$, $S_i$ terminates strongly fair and $S$ does not admit infinite strongly fair execution sequences.

Observe that $R$ is not a free variable of $F$. I.e., for every statement $S$, we define a different $F(R_S)$. Hence, the $F(R)$ are first-order formulae. From now on, we fix some first-order ordinal acceptable structure $\mathfrak{M}$. As before we are able to define the weakest precondition for strongly fair termination sfwp$(S, q)$ for commands $S$ and conditions $q$. Of course, the only interesting case is when $S$ is a repetition. This is the subject of the next theorem.

THEOREM 9.8. *Let* $S \equiv *[\square_{i=1}^{n} b_i \to S_i]$ *with* $n \geqslant 1$. *For every* $\xi$ *the following holds*:

$$\mathfrak{M} \models \mathrm{sfwp}\left(*\left[\mathop{\square}_{i=1}^{n} b_i \to S_i\right], q\right)(\xi)$$

$$\text{iff} \quad \mathfrak{M} \models \left(F(R_S^{sf}) \wedge \left(\left(\bigcup_{i=1}^{n} R_i\right)^* \circ \bigwedge_{i=1}^{n} \neg b_i \to q\right)\right)(\xi).$$

*Proof.* A straightforward adaptation of the proof of Theorem 6.3.4.

COROLLARY 9.9. *Let* $S \equiv *[\square_{i=1}^{n} b_i \to S_i]$. $(n \geqslant 1)$. *For every* $\xi$:

$$\mathfrak{M} \models \mathrm{sfwp}\left(*\left[\mathop{\square}_{i=1}^{n} b_i \to S_i\right], \mathit{true}\right)(\xi) \qquad \mathit{iff} \quad \mathfrak{M} \models F(R_S^{sf})(\xi).$$

Soundness and completeness is established by

THEOREM 9.10. $\mathfrak{M} \models [r]\, S[q]_{sf}$ *iff* $\mathrm{Th}(\mathfrak{M}) \vdash [r]\, S[q]$.

*Proof.* Again, the only non-trivial case is when $S \equiv *[\square_{i=1}^{n} b_i \to S_i]$ with $n \geqslant 1$ holds. The equivalence is proved by induction on the level of $S$.

If $S$ has level 1, i.e., if $S$ has no inner loops, then the theorem follows from the results in Sections 7 and 8. Now suppose that $S$ has level $k+1$ $(k \geqslant 1)$ and that the theorem holds for programs $S$ with level $l$ satisfying $l \leqslant k$. Assume that $\mathfrak{M} \models [r]\, S[q]_{sf}$ holds. Then $\mathfrak{M} \models r \supset [F(R_S^{sf}) \wedge ((\bigcup_{i=1}^{n} R_i)^* \circ \bigwedge_{i=1}^{n} \neg b_i \to q)]$ holds, too. From the definition of $F(R_S^{sf})$, it follows that $\mathfrak{M} \models r \supset [(\bigcup_{i=1}^{n} R_i)^* \to \bigwedge_{i=1}^{n} (b_i \supset F(R_{S_i}^{sf}))]$, i.e., for every execution sequence of $S$ starting in a state satisfying $r$, whenever $b_i$ holds, $S_i$ terminates strongly fair $(i = 1, ..., n)$. For the same reason $\mathfrak{M} \models r \supset \neg \mathrm{SFAIR}(R_1, ..., R_n)$ holds. So, we may proceed as in Section 7 and conclude that $\mathrm{Th}(\mathfrak{M}) \vdash [r]\, S[q]$.

The other implication, i.e., $\mathrm{Th}(\mathfrak{M}) \vdash [r]\, S[q]$ implies $\mathfrak{M} \models [r]\, S[q]_{sf}$, should be obvious.

## 10. Conclusion

We have shown that the $\mu$-calculus can be used as an assertion-language to prove fair termination of do-loops. The notion of fairness considered in this paper is that of strong fairness.

Various rules (Apt *et al.*, 1984; Grümberg *et al.*, 1981; Lehmann *et al.*, 1981; Manna and Pnueli, 1983) for proving strongly fair termination of repetitions have been studied in the literature. All of them have been proved to be sound and complete. However, this was done using set theory as an assertion-language. One of these rules, Orna's rule (Grümberg *et al.*, 1981), is considered in detail in this paper.

The key result of this paper is the fact that the weakest precondition expressing strongly fair termination is definable in the $\mu$-calculus. This result is used in the completeness and soundness proof of the rule. The completeness proof required verifying that the weakest precondition for fair termination implies the premises of the rule. Here, the ordinals are used to define the auxiliary quantities required to apply this rule. We believe that these ordinals can be removed, but we have not done this yet. The soundness proof required to verify that the premises of Orna's rule imply the weakest precondition for fair termination. The LPS-rule (Lehmann *et al.*, 1981), another rule to prove strongly fair termination of do-loops can be shown to be sound and complete in the same manner as Orna's rule.

Future work will be carried out to remove the ordinal constants used in the completeness proof. Furthermore, we will try to define a predicate in the $\mu$-calculus which expresses whether a repetition terminates under the assumption of all-level, i.e., global fairness (Apt *et al.*, 1984). Future research will also be carried out to extend these arguments to more complex forms of fairness and to concurrent programs.

### Acknowledgments

The authors thank P. van Emde Boas, A. Pnueli, and the members of "het Landelijk Seminarium Concurrency," for clarifying remarks, and K. Apt for pointing out an error in an earlier version of this paper. Finally we thank an anynomous referee for his suggestions that led to many improvements, especially regarding the style and notation.

Received June 3, 1986; accepted December 21, 1987

### References

Apt, K. R., and Plotkin, G. D. (1986), "Countable Nondeterminism and Random Assignment," *J. Assoc. Comput. Mach.* 33, No. 4.

Apt, K. R., Pnueli, A., and Stavi, J. (1984), Fair termination revisited—With delay, *Theoret. Comput. Sci.* 33.

DE BAKKER, J. W. (1980), "Mathematical Theory of Program Correctness," Prentice-Hall, Englewood Cliffs, NJ.

DIJKSTRA, E. W. (1976), "A Discipline of Programming," Prentice Hall, Englewood Cliffs, NJ.

GRÜMBERG, O., FRANCEZ, N., MAKOWSKY, J. A., AND DE ROEVER, W. P. (1981), A proof rule for fair termination of guarded commands, *in* "Proceedings, Symposium on Algorithmic Languages," North-Holland, Amsterdam.

HITCHCOCK, P., AND PARK, D. (1973), Induction rules and termination, *in* "Proceedings, ICALP I," North-Holland, Amsterdam.

LEHMANN, D. J., PNUELI, A., AND STAVI, J. (1981), Impartiality, justness and fairness: The ethics of concurrent termination, *in* "Proceedings, ICALP VII," Lecture Notes in Comput. Sci., Vol. 115, Springer-Verlag, New York/Berlin.

MANNA, Z., AND PNUELI, A. (1983), Verification on concurrent programs: A temporal proof-system, *in* "Foundations of Computer Science IV, Part 2," Mathematical Centre Tracts, Vol. 159, Math. Centrum, Amsterdam.

MOSCHOVAKIS, Y. N. (1974), "Elementary Induction on Abstract Structures," North-Holland, Amsterdam.

PARK, D. (1981), A predicate transformer for weak fair iteration, *in* "Proceedings, 6th IBM Symposium on Math. Found. of Computer Science, Hakone, Japan."

PARK, D. (1980), On the semantics of fair parallelism, *in* "Proceedings, Copenhagen Winterschool on Abstract Software Specification, 1979," Lecture Notes in Comput. Sci., Vol. 86, Springer-Verlag, New York/Berlin.

PARK, D. (1969), Fixed point induction and proof of program properties, *Mach. Intell.* 5.

DE ROEVER, W. P. (1981), A formalism for reasoning about fair termination, *in* "Proceedings, Workshop on Programming Logics," Lecture Notes in Comput. Sci., Vol. 131, Springer-Verlag, New York/Berlin.

TARSKI, A. (1955), A lattice-theoretical fixed point theorem and its applications, *Pacific J. Math.* 5.

HAREL, D. (1979), "First-Order Dynamic Logic," Lecture Notes in Comput. Sci., Vol. 68, Springer-Verlag, Berlin/Heidelberg/New York.

SAMENVATTING

Dit proefschrift bestaat uit een bundeling van een viertal artikelen.

De eerste drie artikelen beschrijven een principe voor het ontwerpen van gedistribueerde programma's uit een bepaalde klasse volgens een bijzonder type van redeneren. Deze klasse bestaat uit programma's waarin een groep van knopen in een netwerk een zekere taak uitvoeren die vanuit een logisch oogpunt kan worden opgesplitst in een aantal subtaken alsof deze sequentieel worden uitgevoerd. Vanuit een operationeel oogpunt worden deze subtaken echter concurrent door de knopen uitgevoerd.

Het ontwerpprincipe wordt eerst geïdentificeerd in het eerste artikel, "A correctness proof of a distributed minimum-weight spanning tree algorithm (extended abstract)". Dan wordt in het tweede artikel, "Designing distributed algorithms by means of formal sequentially phased reasoning", een technische formulering van het ontwerpprincipe gegeven. Een toepassing van het principe wordt gegeven in het derde artikel, "A detailed analysis of Gallager, Humblet, and Spira's minimum-weight spanning tree algorithm". In dit artikel worden bovendien twee andere principes geformuleerd: Het eerste beschrijft hoe twee onafhankelijk van elkaar uitgevoerde taken kunnen worden gecombineerd; Het tweede principe is van toepassing wanneer een aantal groepen concurrent ten opzichte van elkaar een aantal taken uitvoeren terwijl een taak uitgevoerd door een groep tijdelijk kan worden verstoord als gevolg van interactie met knopen uit een andere groep.

Het vierde artikel, "The $\mu$-calculus as an assertion-language for fairness arguments", handelt over faire terminatie van do-loops. Hierin wordt de $\mu$-calculus voorgesteld als assertietaal voor het redeneren over dit type van terminatie. Soundness en volledigheid van een regel voor het bewijzen van faire terminatie worden bewezen. Bovendien wordt de zwakste preconditie voor faire terminatie van een do-loop met betrekking tot een zekere postconditie in de $\mu$-calculus gedefinieerd.

Lineaire Temporele Logica (LTL) loopt als een rode draad door de vier bovenstaande artikelen. Wordt het ontwerpprincipe direct geformuleerd met behulp van LTL, in het laatste artikel worden de grondslagen van LTL onderzocht. De resultaten daarvan suggereren (zonder bewijs) dat voor het verifiëren dat een programma fair termineert, een eigenschap die op natuurlijke wijze in LTL geformuleerd kan worden, een assertietaal nodig is die veel meer uitdrukkingskracht heeft dan LTL zelf.

CURRICULUM VITAE

De schrijver van dit proefschrift werd op 22 juli 1957 te Gorssel geboren.
Van 1969 tot 1977 doorliep hij de Thorbecke Scholengemeenschap te Arnhem. Nadat hij het diploma Gymnasium b had behaald begon hij in 1977 met de studie wiskunde aan de Rijksuniversiteit te Utrecht. Zijn kandidaatsexamen wiskunde met bijvak informatica werd in 1981 behaald. Het doctoraalexamen wiskunde met bijvak informatica legde hij in 1984 (cum laude) af.

Sinds 1984 is hij werkzaam bij de afdeling informatica aan de Katholieke Universiteit te Nijmegen (KUN). Daar werkte hij als wetenschappelijk medewerker, eerst in dienst van de KUN, vervolgens in dienst van de Nederlandse Organisatie voor Zuiver-Wetenschappelijk Onderzoek (ZWO). Hij is nu universitair docent in tijdelijke dienst bij de KUN.

CURRENT ADDRESS:

University of Nijmegen,
Department of Computer Science,
Toernooiveld,
6525 ED Nijmegen,
The Netherlands.

**Stellingen**

behorend bij het proefschrift

Design and specification of distributed network algorithms: foundations and applications

van

Frank Stomp

1. Compositionele bewijssystemen, waarbij een specificatie van een programma wordt afgeleid uit specificaties van zijn constituerende programma's zonder te refereren aan de interne structuur van die constituenten [Z89], zijn ongeschikt voor het formaliseren van het soort argumenten dat gebruikt wordt door ontwerpers van netwerk-algoritmen in [GHS83,Hu83,MS79,Se82,Se83,ZS80]. Hiervoor kunnen twee eenvoudige redenen worden gegeven:

   (a) In [GHS83,Hu83,MS79,Se82,Se83,ZS80] worden algoritmen uitgelegd aan de hand van operationele argumenten, waarbij het gebruik van plaatjes ter illustratie niet geschuwd wordt, zonder enige verwijzing naar (de syntactische structuur van) programma's die deze algoritmen beschrijven.

   (b) Ieder compositioneel bewijssysteem legt restricties op aan zijn gebruiker door hem te verbieden gebruik te maken van de interne structuur van een programma, zelfs indien deze wel bekend is.

[GHS83] Gallager R.T., Humblet P.A., and Spira P.M., A distributed algorithm for minimum-weight spanning trees, ACM TOPLAS, 5-1 (1983).

[Hu83] Humblet P.A., A distributed algorithm for minimum-weight directed spanning trees, IEEE Trans. on Comm., 31-6 (1983).

[MS79] Merlin P.M. and Segall A., A failsafe distributed routing protocol, IEEE Trans. on Comm., 27-9 (1979).

[Se82] Segall A., Decentralized maximum-flow algorithms, Networks 12 (1982).

[Se83] Segall A., Distributed network protocols, IEEE Trans. on Inf. Theory. IT29-1 (1983).

[Z89] Zwiers J., Compositionality, concurrency, and partial correctness: Proof theories for networks of

processes, and their connection, Ph. D. Thesis, Eindhoven University of Technology (1988). (Ook verschenen als LNCS 231 (1989)).

[ZS80] Zerbib F.B.M. and Segall A., A distributed shortest path protocol, Internal Report EE-395, Technion-Israel Institute of Technology, Haifa, Israel (1980).

2. Een principe voor het ontwerpen van failsafe algoritmen [S89] kan in dezelfde trant geformuleerd worden als het principe voor het sequentieel redeneren over concurrent uitgevoerde subtaken (zie hoofdstuk 3 van dit proefschrift). Een dergelijk principe biedt de mogelijkheid na te gaan of de algoritmen in [SH86] inderdaad correct zijn. Bovendien kan het tot een beter inzicht leiden in de onjuistheid, aangetoond in [SH86], van een van Finn's algoritmen [F79].

[F79] Finn S.G., Resynch procedures and a failsafe network protocol, IEEE Trans. on Comm., Vol. COM-27 (1979).

[SH86] Soloway S.R. and Humblet P.A., On distributed network protocols for changing topologies, Technical Report LIDS-P-1564, MIT (1986).

[S89] Stomp F.A., A principle for formally designing failsafe algorithms, in voorbereiding.

3. De bewering van Chou en Gafni dat zij in [CG88] de correctheid van Gallager, Humblet en Spira's algoritme [GHS83] Gallager's algoritme– bewezen hebben is onjuist.

Chou en Gafni beschouwen een gedistribueerde implementatie van Boruvka's algoritme [B26] (ook te beschouwen als een gedistribueerde implementatie van Kruskal's algoritme [K56]), waarin twee groepen van knopen alleen kunnen worden gecombineerd tot één groep als hun minimale uitgaande kanten dezelfde zijn.

In tegenstelling tot Gallager's algoritme, kunnen in het algoritme dat in [CG88] geanalyseerd wordt geen combinaties van groepen plaatsvinden indien hun minimale uitgaande kanten verschillend zijn. Juist dit type van combinaties is het karakteristieke van Gallager's algoritme en vereist een complexere analyse dan die in [CG88].

[B26] Borovka O., O jistém problému minimálním. Práca Moravské Přírodovědecké Společnosti (1926) (in Czech.).

[CG88] Chou C.T. and Gafni E., Understanding and verifying distributed algorithms using stratified decomposition, Proc. of the ACM Symp. on Principles of Distr. Comp. (1988).

[K56] Kruskal J.B., On the shortest spanning subtree of a graph and the traveling salesman problem, Proc. Am. Math. Soc., 7 (1956).

[GHS83]: zie stelling 1.

4. Het pattern matching probleem vraagt naar het meest linker voorkomen van een patroon in een zekere tekst. Een eenvoudige oplossing hiervan bestaat uit het stapsgewijs van links naar rechts doorlopen van de tekst op zoek naar het patroon.

   Een ingenieuzere methode wordt door Boyer en Moore in [BM77] gegeven. Hierbij wordt gebruik gemaakt van het feit dat het in het algemeen mogelijk is het patroon meer dat één positie in de tekst naar rechts te verschuiven indien het patroon nog niet in de tekst herkend is. Een formele afleiding van Boyer en Moore's algoritme wordt in [PS89a] gegeven.

   Het aantal posities, dat afhangt van (een deel van) het patroon en van een karakter (zie [BM77]), kan volgens Boyer en Moore in een tabel worden opgeslagen die berekend wordt voordat het eigenlijke algoritme wordt uitgevoerd ("preprocessing"). Deze berekening is zelf echter weer een instantie van het pattern matching probleem.

   Dit laatste wordt in [BM77] gespecificeerd maar niet algoritmisch opgelost. Een algoritmische oplossing van dit probleem wordt in [KMP77] gegeven, maar in [R80] wordt aangetoond dat het algoritme in [KMP77] niet aan de specificatie van [BM77] voldoet.

   Door middel van een geschikte generalizatie van de afleiding in [PS89a] kan een volledige en correcte operationele oplossing van het preprocessing probleem formeel worden afgeleid, zie [PS89b].

   [BM77] Boyer R.S. and Moore J.S., A fast string searching algorithm. Comm. ACM. 20-10 (1977).

   [KMP77] Knuth D.E., Morris J.H., and Pratt V.B., Fast pattern matching in strings, SIAM. J. Comput. 6 (1977).

   [PS89a] Partsch H.A. en Stomp F.A., A fast pattern matching algorithm derived by transformational and assertional reasoning, submitted for publication (1989).

   [PS89b] Partsch H.A. en Stomp F.A., Reusability through generalization, in voorbereiding.

   [R80] Rytter W., A correct preprocessing algorithm for Boyer-Moore string searching, SIAM. J. Comput. 9 (1980).

5. Transformationeel programmeren is zeer geschikt voor het analyseren van netwerk-algoritmen zoals beschreven in [GHS83,Hu83,MS79,Se82,Se83,ZS80] omdat

   - reeds op een non-implementeerbaar niveau van beschrijving de essentie van een algoritme begrepen kan worden,
   - de veelal uitstekende (informele) uitleg van de ontwerper direct geformaliseerd kan worden en
   - essentiële beslissingen van de ontwerper geïdentificeerd, geverifiëerd en kwalitatief geanalyseerd kunnen worden tijdens de ontwerpfase.

   [GHS83], [Hu83], [MS79], [Se82], [Se83] en [ZS80]: zie stelling 1.

6. Manna en Pnueli reduceren in [MP83] een bewijs van een temporele eigenschap van een programma tot een bewijs van eigenschappen van de transities van dat programma. De suggestie dat voor het redeneren over dergelijke transities Lineaire Temporele Logica, zoals gedefiniëerd in [MP83], een voldoende expressieve taal is, is onjuist. Een formalisme dat krachtiger is dan Lineaire Temporele Logica, bijvoorbeeld de $\mu$-calculus, is hiervoor vereist. Zie hoofdstuk 5 van dit proefschrift, en ook [G84].

[G84] Gerth R.T., Transition Logic: How to prove temporal properties in a compositional way, Proc. 16th ACM Symposium on the Theory of Computing (1984).

[MP83] Manna Z. and Pnueli A., Verification of concurrent programs: A temporal proof system, Foundations of Computer Science IV, part 2, Mathematical Centre tracts 159 (1983).

7. Hoewel het typische ritme van reggae-muziek de liefhebber in vervoering kan brengen, bepaalt juist de gezongen of gesproken tekst in sterke mate het karakter van deze muziek.