The Verified Incremental Design of a Distributed Spanning Tree Algorithm

Hesselink, Wim H.

Formal Aspects of Computing (1999) 11: 45–55
© 1999 BCS

Formal Aspects of Computing

The Verified Incremental Design of a Distributed Spanning Tree Algorithm: Extended Abstract

Wim H. Hesselink

Department of Mathematics and Computing Science, Rijksuniversiteit Groningen, Groningen, The Netherlands

Keywords: Minimum spanning tree; Message passing; Asynchronous communication; Theorem proving

Abstract. The paper announces an incremental mechanically–verified design of the algorithm of Gallager, Humblet, and Spira for the distributed determination of the minimum-weight spanning tree in a graph of processes. The processes communicate by means of asynchronous messages with their neighbours in the graph. Messages over one link may pass each other. The proof of the algorithm is based on ghost variables, invariants, and a decreasing variant function. The verification is mechanized by means of the theorem prover Nqthm of Boyer and Moore. This extended abstract is an introduction to the full paper that can be obtained by ftp (http://link.springer.de/link/service/journals/00165/).

1. Introduction

Given is a connected undirected graph in which all edges have different weights. So this graph has a unique minimum–weight spanning tree. The nodes of the graph are processes that can asynchronously send messages to neighbour processes. Every process only knows the weights of its incident edges and the names of its neighbours.

In 1983, Gallager, Humblet, and Spira published a distributed algorithm for these processes to determine the minimum–weight spanning tree, cf. [GHS83]. The algorithm is not very hard to understand and there are good informal explanations, e.g. cf. [Tel94]. The level of concurrency allowed, however, makes it hard to verify that no undesired interference or deadlock occurs. The correctness has been verified by a number of groups, cf. [ChG88, SdR94, WLL88, ZwJ93], but handwritten proofs are almost never complete and hence not very convincing.

Correspondence and offprint requests to: Wim H. Hesselink, Department of Mathematics and Computing Science, Rijksuniversiteit Groningen, Postbox 800, 9700 AV Groningen, The Netherlands. email: wim@cs.rug.nl

We have therefore undertaken the construction of a proof for a mechanical theorem prover, so that anyone who understands the language of the prover can verify the proof, i.e., can see what it is we are asserting, can let the prover verify the assertions, and can inspect any detail they want to look into. We did not prove the precise algorithm of [GHS83], but our algorithm is closely related and at least as efficient.

Some earlier proofs, cf. [SdR94, ZwJ93], started with the verification of a sequential program, which is then gradually distributed in a number of program transformations. In contrast to this approach, we start with a highly nondeterministic distributed algorithm which is gradually tuned to fulfil the specification. We use a kind of reverse engineering. Knowing the algorithm of [GHS83], we perform a verified incremental design of it. So in each stage of the project, we know the invariant properties of the algorithm at that stage.

A full description of the design and the proof requires more than 60 pages. This length combined with the specialized nature of the material is prohibitive for most methods of publication. We have therefore chosen for electronic publication in [Hes98]. Here we only give an overview of the algorithm and some highlights of the proof. The complete mechanical proof is available on WWW, see [Hes@]. Our approach to such problems is described in a simpler setting in [Hes97].

2. Formalization

We let the graph consist of a set V of nodes and a set E of edges (ordered pairs of nodes). The edges have weights given by a function $w \in E \to \mathbb{R}$. To eliminate E, we extend w to a function $V \times V \to \mathbb{R} \cup \{\infty\}$ by defining $w.(x,y) = \infty$ iff $(x,y) \notin E$. For all nodes x and y, we have $w.(x,y) = w.(y,x)$ and $w.(x,x) = \infty$, since the graph is undirected and without selfloops. We postulate that all finite weights are different and that the graph is connected.

It is well known that under these circumstances the minimum-weight spanning tree is unique. We regard it as a binary relation on V and call it *MST*.

The algorithm is a distributed version of Boruvka's algorithm, cf. [Tar83]. Relation *MST* is determined by means of the following result. Let an edge (x,y) be called an outgoing edge of a set C of nodes iff $x \in C$ and $y \notin C$. A lightest outgoing edge of C is an outgoing edge of C with the smallest weight. It is now easy to verify

Theorem. The lightest outgoing edge of any set of nodes belongs to *MST*.

This result is used in the algorithm to construct *MST* as a growing forest F. So we introduce the invariant $F \subseteq MST$. Boruvka's algorithm uses the Theorem with a component of forest F for the set of nodes. Let F^* be the reflexive transitive closure of relation F. The algorithm is given by

```
$F := \emptyset$ ; $going := true$ ;
while $going$ do
    $choose\ v \in V$ ;
    $C := \{x \mid (v,x) \in F^*\}$ ;
    if $possible$ let $(x,y)$ be
            $the\ lightest\ outgoing\ edge\ of\ C$ ;
        $F := F \cup \{(x,y),(y,x)\}$
    else $going := false$
od
```

Upon termination, the set C is nonempty and has no outgoing edges. Since (V, E) is connected, it follows that C equals V, so that $(v, x) \in F^*$ for all nodes x. Since MST is a tree and $F \subseteq MST$, this implies $F = MST$.

In the distributed algorithm the nodes of the graph are processes that communicate by means of asynchronous messages to neighbour nodes. We assume that, initially, one *wakeup* message is in transit to every process (sent but not yet accepted) and that there are no other messages. The model does not guarantee that this is the first message to be accepted.

Forest F is distributed over the graph by providing every node q with a set-valued private variable $branch^+.q$. The forest is then given by $(q, r) \in F$ iff $r \in branch^+.q$ (and $r \neq q$).

We provide every node p with a boolean variable $term.p$ for termination detection. The goal of the algorithm is that, if any node knows termination, then MST is known everywhere, i.e., for all nodes p, q, r:

$$term.p \quad \Rightarrow \quad ((q, r) \in MST \equiv r \in branch^+.q) \tag{1}$$

It is also required that after a finite number of accepted messages every process p satisfies $term.p$ and there are no more messages in transit.

This specification differs slightly from the description in [GHS83], where processes can wake up spontaneously or upon receiving messages from awakened neighbours. Moreover, we treat termination detection more explicitly.

3. Modelling Asynchrony

We need to go into the modelling assumptions. Every process has a private state consisting of a number of private variables. Processes can send messages to neighbour processes. A process acts only when it accepts a message. Every message has a key word and a number of arguments. Via the declaration of the algorithm, the key word and the arguments determine the enabling condition of the message and the associated command. Acceptance of a message consists of its removal from the network together with the execution of its command. The enabling condition is the precondition for acceptance. The command can only inspect and modify private variables and send messages to neighbour processes; it always terminates. All processes concurrently execute the sequential program

```
while true do accept some enabled message or wait od
```

The only fairness assumption is that, whenever some enabled messages exist, one will be accepted eventually.

In the physical model the acceptance of a message takes some time, and message acceptances of different processes may overlap. The possibility and the effect of acceptance, however, only depend on the message and the private state of the accepting process prior to acceptance. Since the acceptance is finished before the process can accept a next message and since the sending of messages only adds them to the bag of messages in transit, we may regard the acceptance of a message as a single atomic action and we may regard the atomic actions as interleaved.

In this way we arrive at the following mathematical model, a simple version of the model of [Tel94]. The *state* of the system consists of the private states of the processes together with the bag of messages that are in transit. A *transition* is a step from one state to another in which a single process accepts an enabled message. An *execution* of the algorithm is a sequence of transitions that starts in some initial state. A state is called *reachable* if it occurs in an execution.

Since in every step of the system only one process is involved, this model of concurrency is simpler than the models for synchronous communication.

3.1. Invariants

An *invariant* is defined to be a predicate that holds in all reachable states. We write $P \rhd Q$ to denote that every atomic step of the algorithm that starts in a state where P holds, terminates in a state where Q holds. We define a predicate P to be a *strong invariant* if it holds initially and satisfies $P \rhd P$. It is easy to see that every predicate implied by a strong invariant is an invariant. The usual way to obtain (strong) invariants is based on the following obvious result.

Theorem. Let Q be the conjunction of a family of predicates P_i with $i \in I$. Assume that Q holds initially and that $Q \rhd P_i$ for all $i \in I$. Then Q is a strong invariant.

In such a situation, the predicates P_i follow from Q and are therefore invariants. Since they are used to construct a strong invariant, they are called *constituent invariants*. For GHS we need a family of some 160 constituent invariants. So the main effort in the design was to manage this host of invariants.

3.2. Messages in Transit

When we began to investigate the GHS algorithm, we had no idea what kind of invariants to use. Since messages are transient, we did not expect them in invariants. This turned out to be mistaken. For, in the end, most invariants express a property of a node when a certain message is in transit to it or from it.

For the formal description of the global state, we introduce variables $buf.q$ to hold the bag of messages in transit to process q. So, if process p sends a message with key word kw and arguments a to q, according to the command $send(q, kw, a)$, this has the effect

$$buf.q := buf.q + \{(kw, a)\}$$

where $+$ denotes bag addition. Process q can accept any enabled message $m \in buf.q$. Acceptance of m has the effect that m is removed from $buf.q$ and that the body of message m is executed.

We use two other sending commands. Firstly, a multicast to a set of destinations is expressed by $mcast(S, kw, a)$, which is equivalent to

$$\textbf{for all } r \in S \textbf{ do } send(r, kw, a) \textbf{ od}$$

Secondly, we introduce the possibility that a process sends a message m to itself by means of the command $delay(m)$. The purpose of selfmessages is to postpone an action until execution is appropriate.

In order to discuss the messages in transit, we introduce notations like

$$(kw, j) \textbf{ at } q \quad \equiv \quad (\exists\, b :: (kw, j, b) \in buf.q)$$

which express that some message is in transit to q with key word kw and first argument j.

Remarks. In contrast to [GHS83], we assume point to point communication: processes send messages to neighbour nodes and edges are merely pairs of nodes (the same is done in [WLL88]). In this way, we avoid channel names, but we disallow multiple edges.

We do not assume preservation of the order of messages sent along one edge. This makes the algorithm of [GHS83] incorrect, see Section 7. Moreover, instead of disabling, the processes of [GHS83] may have to put the message back at the end of the message queue. Then the order of the queue is a complicating factor.

4. The Bottom Layer of the Design

The final design contains declarations of eleven messages, but only three messages modify the growing forest as represented by $branch^+$. In this Section, we present simple forms of these messages and some associated invariants.

For the sake of the algorithm, we break the apparent symmetry of the forest. This is done by giving each node a private variable *ib* (standing for *in-branch*) and a set-valued variable *branch* such that $branch^+.q = \{ib.q\} \cup branch.q$. So, now $branch^+$ is a "derived" variable. The variables *ib* define the structure of a directed graph in which every node q has precisely one "parent" $ib.q$. If two nodes are parent to each other, we speak of a *core*.

The relations between *ib* and *branch* are expressed in the invariants:

(J0) $\quad q \in branch.r \;\Rightarrow\; ib.q = r$

(J1) $\quad ib.q \notin branch.q$

In view of (J0), $branch.r$ may be regarded as a set of children of r. Since $branch^+.q$ and $branch^+.r$ cannot be modified synchronously, we have to reckon with the situation

$$r \in branch^+.q \quad \wedge \quad q \notin branch^+.r$$

in which case $r = ib.q$ because of (J0). In this situation, we require that a message $(connect, q)$ is in transit to node r. So we postulate the invariant

(J2) $\quad q \in branch.(ib.q) \;\vee\; ib.(ib.q) = q \;\vee\; (connect, q)\ \mathbf{at}\ ib.q$

Initially, every node q has $ib.q = q$ and a message *wakeup* is in transit to it. Moreover, we assume that the private variable $be.q$ then holds the nearest neighbour of q (*be* stands for *best-edge*). Apart from some minor details, message *wakeup* is declared by:

```
accept (wakeup) =
•  if  ib = self  then
        ib := be ;
        send (be, connect, self)
   fi
end
```

The bullet serves to separate the enabling condition from the command, but we omit the enabling condition if the message is always enabled. So, if node q has a *wakeup* message, it can always accept it. Acceptance means removal from $buf.q$ and execution of the body. For *wakeup*, the body does nothing if $ib.q \neq q$. If $ib.q = q$, then $ib.q$ becomes $be.q$ and a message $(connect, q)$ is sent to $be.q$.

In order to preserve (J2) when node *ib.q* accepts the message (*connect*, *q*), the first approximation of *connect* is

```
**accept** $(connect, j) =$
• **if** $j \neq ib$ **then** $branch := branch \cup \{j\}$ **fi**
**end**
```

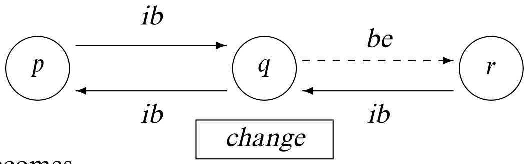
Now the question arises how to extend the forest. For this purpose, we use the second message that modifies *ib*. It is approximately given by

```
**accept** $(change) =$
• **if** $be \in branch$ **then** $send\,(be, change)$
  **else** $send\,(be, connect, self)$ **fi** ;
  $branch := (branch \cup \{ib\}) \setminus \{be\}$ ;
  $ib := be$
**end**
```

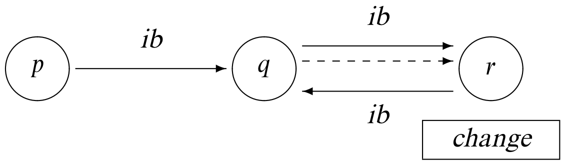
In order not to violate (J0), we postulate that a *change* message is always at a core, as expressed in the invariant

(J3) $\quad change$ **at** $q \quad \Rightarrow \quad ib.q \neq q \quad \wedge \quad ib.(ib.q) = q \;.$

According to the declaration of *change*, execution of *change* by *q* destroys this core by resetting *ib.q*. If $be.q \in branch.q$, it follows from (J0) that $(q, be.q)$ becomes a new core with a new *change* message **at** *be.q*. If we put $p = ib.q$ and $r = be.q$, the inital situation is



and it becomes



So, the message *change* pulls the core along the path of *be*–arrows. This will be used in other layers to move the core to the lightest outgoing edge of the component. On the other hand, if $be.q \notin branch.q$, then the core dissolves and a *connect* message is sent. Other layers will have to guarantee that this occurs at the lightest outgoing edge.

The invariant (J3) has the drawback that in later applications it is not clear which of the two consequents is relevant. We therefore split it into a conjunction of two invariants in the obvious way. In the effort to prove the invariance of these predicates, we end with fourteen invariants, which form the bottom layer of the design. This bottom layer contains all modifications of the private variables *ib* and *branch*. The remainder of the algorithm is concerned with the value of *be* and the emergence of *change* messages. In the subsequent layers the declarations of *wakeup* and *change* are extended slightly, but the declaration of *connect* grows to more than half a page.

5. The Remainder of the Design

According to the algorithm of Section 2, the nodes of a component of the growing forest have to find the lightest outgoing edge of the component and to add this edge to the forest, until no such edges can be found. To decide that an edge is outgoing, every node must know the component it belongs to. In the algorithm, this component information is generated when two nodes form a new core by sending *connect* messages to each other. The information is then broadcast over the component by means of *init* messages. Thus, we build a layer with *init* messages and component information upon the bottom layer of Section 4.

In the next layer, every node of a component uses messages *search*, *ask*, *answer* to investigate its own edges. Then the results of the local searches are reported to the core by means of messages *sendrep* and *report*. In the next layer the two core nodes must agree on the choice of a lightest outgoing edge. Then the core moves to this edge by means of a chain of *change* messages. If no outgoing edge is found, *halt* messages are spread and the algorithm terminates.

At this point, the algorithm is conditionally correct, but it may end in deadlock, since a component can join another component without generating a new core. The remedy is a new layer with *winit* messages (weak init) to distribute lacking component information. In a final layer, we construct a decreasing variant function to prove termination and to give a bound on the message complexity of the algorithm.

The aim of each new layer is to fulfil some aspect of the design. This always necessitates new invariants. We sometimes add new messages or new actions on variables. In that case, we usually also need new invariants to prove that the established invariants are preserved.

In some cases during the design, we had to abolish an invariant that blocked the design. Of course we tried to avoid such backtracking as much as possible. We have been guided in our intuitions about invariants by the existing algorithm of [GHS83], even though almost no invariants of that algorithm are known.

We use four private variables in the design that do not occur in the final algorithm. The boolean variable *fnd.q* expresses that node q and its children have to find a good outgoing edge. The boolean variable *srch.q* expresses that q is to search its neighbours. The variable *explist.q* holds the set of neighbours node q is expecting *report* messages from. These variables are made superfluous by the introduction of an integer variable *fc* with the invariant

$$fc.q \quad = \quad \#fnd.q + \#srch.q + \#explist.q$$

where, for P boolean, $\#P$ is 1 if P holds and 0 otherwise, and for S a set, $\#S$ is the number of elements of S. After the introduction of *fc* the variables *fnd*, *srch*, and *explist* are so-called ghost variables, useful for the proof but redundant in the algorithm. Our variable *fc* is a variation of the variable *find-count* of [GHS83], which counts different things. Finally, the variant function uses a ghost variable *bash*, which differs slightly from the actual variable *bas*. We have removed these ghost variables from the algorithm in the final stage of the mechanical proof.

6. The Algorithm without Ghost Variables

The design sketched above is presented in [Hes98]. It results in the following algorithm. Each process has eleven private variables

```
$ib, be, te : node$ ;
$term, mar : boolean$ ;
$branch, bas$ : set of $node$ ;
$ll, bw, fc, ci : number$ .
```

We use functions *lewe* and *lenb* for least weight and least neighbour of a node q with respect to a set of nodes S. These functions are defined as follows. If there is a node $r \in S$ with $w.(q,r) < \infty$ and $w.(q,r) \leqslant w.(q,x)$ for all $x \in S$ then $lewe.(q,S) = w.(q,r)$ and $lenb.(q,S) = r$. Otherwise $lewe.(q,S) = \infty$ and $lenb.(q,S) = q$.

We postulate the initial conditions:

$$\begin{aligned} & ib.q = q \quad \wedge \quad te.q = q \quad \wedge \quad branch.q = \emptyset \quad \wedge \quad ll.q = 0 \\ \wedge \quad & \neg term.q \quad \wedge \quad \neg mar.q \quad \wedge \quad fc.q = 0 \quad \wedge \quad bas.q = \{r \mid w.(q,r) < \infty\} \\ \wedge \quad & bw.q = lewe.(q,V) < \infty \quad \wedge \quad be.q = lenb.(q,V) \\ \wedge \quad & buf.q = \{(wakeup)\} \end{aligned}$$

The bodies of the messages *connect* and *init* use a procedure *initp*, given by

```
proc  $initp\ (v, id)$ =
    $ll := v$ ; $ci := id$ ; $be := ib$ ; $bw := \infty$ ;
    $fc := \#branch + 2$ ;
    $delay\ (sendrep)$ ; $delay\ (search)$ ;
    $mcast\ (branch, init, v, id)$
end
```

The eleven messages are declared by

```
accept $(wakeup)$ =
• if $ib = self$ then
      $ib := be$ ;
      $bas := bas \setminus \{be\}$ ;
      $send\ (be, connect, self, ll)$
  fi
end .

accept $(init, v, id)$ =
  enabling $\neg mar$
• $initp\ (v, id)$
end .

accept $(sendrep)$ =
  enabling $fc = 1$
• $fc := 0$ ;
  $send\ (ib, report, self, bw)$
end .

accept $(search)$ =
• if $lewe.(self, bas) < bw$ then
      $te := lenb.(self, bas)$ ;
      $send\ (te, ask, self, ll, ci)$
  else $fc := fc - 1$ fi
end .
```

```
accept $(connect, j, v)$ =
  enabling $j = ib \vee v < ll$
• if $j = ib$ then
      $mar := true$ ;
      $initp(ll + 1, w.(self, j))$
  else
      $branch := branch \cup \{j\}$ ;
      $bas := bas \setminus \{j\}$ ;
      if $w.(j, self) < bw$ then
          $send\ (j, init, ll, ci)$ ;
          $fc := fc + 1$
      else $send\ (j, winit, ll, ci)$ fi
  fi
end .

accept $(ask, j, v, id)$ =
  enabling $v \leqslant ll$
• if $ci \neq id$ then
      $send\ (j, answer, false)$
  else
      $bas := bas \setminus \{j\}$ ;
      if $j = te$ then
          $te := self$ ;
          $delay\ (search)$
      else $send\ (j, answer, true)$ fi
  fi
end .
```

```
accept (answer, b) =
• if $b$ then
      $bas := bas \setminus \{te\}$ ;
      delay (search)
  else
      $fc := fc - 1$ ;
      if $w.(self, te) < bw$ then
         $be := te$ ; $bw := w.(self, te)$
      fi
  fi ;
  $te := self$ ;
end .

accept (winit, v, id) =
• if $ll < v$ then
      $ll := v$ ; $ci := id$ ; $be := ib$ ;
      mcast (branch, winit, v, id)
  fi
end .

accept (halt) =
• $term := true$ ;
  mcast(branch, halt)
end .

accept (report, j, v) =
  enabling $j \neq ib \lor (mar \land fc = 0)$
• if $j \neq ib$ then
      $fc := fc - 1$ ;
      if $v < bw$ then
         $be := j$ ; $bw := v$
      fi
  else
      $mar := false$ ;
      if $bw < v$ then delay (change)
      elsif $v = \infty$ then delay (halt) fi
  fi
end .

accept (change) =
• if $be \in branch$ then
      send (be, change)
  else send (be, connect, self, ll) fi ;
  $branch := (branch \cup \{ib\}) \setminus \{be\}$ ;
  $bas := bas \setminus \{be\}$ ;
  $ib := be$
end .
```

7. Differences with GHS

Conceptually, our algorithm is a version of the algorithm of [GHS83]. Yet there are several details where it differs. We only sketch the more important ones.

Firstly, in [GHS83], the variable *in-branch*, which is our *ib*, is reset by *Initiate* (our message *init*), and not by *Change-root* (our message *change*) as in our version. Note however that on page 72 of [GHS83] a message *Change-core* is said to have the effect that "the inbound edge … is changed to correspond to *best-edge*". Secondly, the handshake that forms a new core requires two *Initiate* messages not needed in our version. These two deviations from [GHS83] make it hard to adapt our proof to the version of [GHS83].

Our selfmessage *search* is an optimized version of procedure *test* of [GHS83]: a node p only sends an *ask* message to neighbour q when the weight of the edge is less than $bw.p$. This applies when node p has obtained a small value of bw by a *report* from one of its children. We have a similar optimization in the **else** part of *connect*, where an *init* message is only sent if the relevant edge has a weight less than $bw.p$. These modifications do not influence the estimates for the worst case complexity.

At three points the algorithm of [GHS83] needs fifo channels, although Tel ([Tel94], pp. 67, 244) suggests otherwise. The first point is that, when a new core is formed, the *Initiate* message must not be passed by the *Report* message that may dissolve the core. Secondly, such a dissolving *Report* message must not be passed by a new *Initiate* message. Thirdly, different *Initiate* messages must not pass each other. Since we do not require fifo channels, we have to avert these dangers by other means. Our boolean variable *mar* serves for the first two points. The third point is solved by the condition $ll < v$ in message *winit*.

8. Mechanical Verification

The verifications of the bottom layer (see Section 4) can be done by hand, but for the full algorithm manual verification is unfeasible. We therefore use the mechanical theorem prover Nqthm of [BoM88] for this purpose.

The best way to construct a mechanical proof is to start with a virtually complete handwritten proof. In the present project this ideal was not realizable, since the proof is too large. For each of the thirteen layers, however, it was possible to start with a handwritten proof.

In the proof, we construct `(ghs0 n ora g)` as the final state when the algorithm starts in some initial state and takes `n` steps in graph g, where a step is the acceptance of an enabled message if there is one and *skip* otherwise. The nondeterminacy in the initial state and in the steps is modelled by the oracle `ora`. We use a predicate `goodgraph` to capture the conditions on graph `g`: it must be connected and all finite weights must be different.

All proof obligations mentioned in Section 2 are met in our mechanical proof, see [Hes98]. Let us here only give the simplified assertion that, after sufficiently many steps of the algorithm we have $(q,r) \in MST \equiv r \in branch^{+}.q$. This is contained in the NQTHM lemma

```
(lemma ghs0-leads-to-goal (rewrite)
    (implies (and (member q (nodes g))
                  (member r (nodes g))
                  (not (lessp n (vfstart g (nodes g))))
                  (goodgraph g) )
             (iff (mintree g q r)
                  (member r (branch+ q
                                     (ghs0 n ora g) )) ) ) )
```

The function *vfstart* used here expresses termination, but also gives a good estimate for the message complexity of the algorithm. In fact, its value is an upper bound for the number of messages that are sent and accepted during the algorithm. If n is the number of nodes and e is the number of edges, then *vfstart* equals $4e - 3n + (7n - 2)L$, where L is the number of binary digits of n. Not counting the selfmessages *search* and *sendrep* we get the estimate $2e{-}n{+}(5n{-}2)L$, cf. [GHS83].

9. Conclusions

Redesign of the algorithm provided motivation for almost all design decisions of [GHS83]. We were able to add some minor optimizations, without making the proof more complex. The grain of atomicity has been made somewhat finer by the introduction of the selfmessages *search*, *sendrep*, and at one point *change* and *halt*.

Early in the design we decided that fifo channels should not be needed for the algorithm and would complicate the proof unnecessarily. This guess turned out to be justified. Although the original version of [GHS83] needs fifo channels, the fifo assumption has been removed rather easily, see Section 7.

The proof techniques used are completely classical: ghost variables, invariants, and variant functions for termination. They were combined with the use of a powerful first-order theorem prover for book-keeping. The proof required much work, more or less quadratic in the number of invariants. For, with every extension, we had to go through all previous invariants. In many cases, the theorem prover decided that no new arguments were needed, but usually there was a fraction that needed additional arguments.

References

[BoM88] Boyer, R. S. and Moore, J.: *A Computational Logic Handbook*. Academic Press, Boston, 1988.

[ChG88] Chou, C. and Gafni, E.: Understanding and verifying distributed algorithms using stratified decomposition. In *Proceedings 7th ACM Symposium on Principles of Distributed Computing*, 1988.

[GHS83] Gallager, R. G., Humblet, P. A. and Spira, P. M.: A distributed algorithm for minimum–weight spanning trees. *ACM Trans. on Programming Languages and Systems*, **5**: 66–77, 1983.

[Hes97] Hesselink, W. H.: A mechanical proof of Segall's PIF algorithm. *Formal Aspects of Computing*, 9: 208–226, 1997.

[Hes98] Hesselink, W. H.: The verified incremental design of a distributed spanning tree algorithm. *Formal Aspects of Computing*, 11(E).
Web site: http://link.springer.de/link/service/journals/00165/.

[Hes@] Hesselink, W. H.: Web site: http://www.cs.rug.nl/~wim/ghs

[SdR94] Stomp, F. A. and Roever, W.-P. de: Principles for sequential reasoning about distributed algorithms. *Formal Aspects of Computing*, 6(E), pp 1–70 (1994). Can be retrieved by downloading the file FACj-6E-p1.ps.Z in directory pub/fac of ftp.cs.man.ac.uk.

[Tar83] Tarjan, R. E.: Data structures and network algorithms. Society for Industrial and Applied Mathematics 1983.

[Tel94] Tel. G.: *Distributed Algorithms*. Cambridge University Press, 1994.

[WLL88] Welch, J., Lamport, L. and Lynch, N.: A lattice-structured proof technique applied to a minimum weight spanning tree algorithm. In *Proceedings 7th ACM Symposium on Principles of Distributed Computing*, 1988, pp. 28–43.

[ZwJ93] Zwiers, J. and Janssen, W.: Partial order based design of concurrent systems. In J. de Bakker, W.-P. de Roever, and G. Rozenberg (eds.): *Proceedings of the REX School/Symposium "A decade of concurrency", Noordwijkerhout, 1993, LNCS 803*, Springer Verlag, 1994, pp. 622–684.

Received May 1997
Accepted in revised form January 1995