Distrib. Comput. (2001) 14: 31–39

DISTRIBUTED COMPUTING

Wait-free concurrent memory management by Create and Read until Deletion (CaRuD)

Wim H. Hesselink[1], **Jan Friso Groote**[2]

[1] Deptartment of Mathematics and Computing Science, University of Groningen, P. O. Box 800, 9700 AV Groningen, The Netherlands (e-mail: wim@cs. rug. nl)
[2] CWI, P. O. Box 94079, 1090 GB Amsterdam, The Netherlands and Department of Mathematics and Computing Science, Eindhoven University of Technology, P. O. Box 513, 5600 MB Eindhoven, The Netherlands (e-mail: jfg@cwi. nl)

Received: July 1998 / Accepted: July 2000

Summary. The acronym CaRuD represents an interface specification and an algorithm for the management of memory shared by concurrent processes. The memory cells form a directed acyclic graph. This graph is only modified by adding a new node with a list of reachable children, and by removing unreachable nodes. If memory is not full, the algorithm ensures wait-free redistribution of free nodes. It uses atomic counters for reference counting and consensus variables to ensure exclusive access. Performance is enhanced by using nondeterminacy guided by insecure knowledge. Experiments indicate that the algorithm is very suitable for multiprocessing.

Key words: Concurrent garbage collection – Reference counting – Shared memory – Wait-free – Consensus – Terms

1 Introduction

The setting of this note is a system of concurrent sequential processes that operate on a common database of terms, such as Lisp's S-expressions or annotated terms as in [2]; see [2] for an overview of term formats. Terms are to be understood in their normal mathematical sense. So, typical terms are $2 + x$, $f(a, x, y, g(a, b))$. Terms are very useful and well-known objects for which many theories and tools are available for rewriting, unification, automated reasoning, etc. Indeed, terms can be used as the primary objects to be exchanged between tools. This is shown, e. g. , in the software coordination architecture of Bergstra and Klop [1]. The architecture we propose can also be used for concurrent implementations of functional languages and theorem provers.

From an implementation point of view, terms are simply directed acyclic graphs where each node is labelled with a function name. It turns out that terms can be efficiently used even when we restrict the number of operations on them as follows. A term can be created once and inspected as long as needed. Terms that are not in use any longer can be deleted by a garbage collector. Viewed in this way, creation and deletion of terms is a memory management problem.

The restriction to these operations on terms has two advantages. First, it allows sharing of subterms, reducing memory requirements substantially. Second, it allows parallel access to terms, since terms are basically static objects in memory. Of course, nothing comes for free. Since terms cannot be changed, a common operation such as the substitution of a term for a variable must be carried out by copying the root path of the variable and providing a pointer to the term.

Some implementations of the CaRuD interface exist, e. g. , [2,1]. These all assume sequential creation, access and deletion of terms. The CaRuD architecture, however, makes it possible that terms are accessed concurrently within shared memory environments. We found that the use of synchronisation primitives to guarantee exclusive access is relatively slow and does not scale up to more processors [14]. Therefore, there is reason to look for a wait-free solution in which a process that needs a new node gets one within bounded delay, independently of actions of other processes, cf. [8]. Since several processes may be contending for the same node, consensus is needed to decide which process succeeds. Consensus can be forced by delegating redistribution to a central garbage collector. We prefer not to create this bottleneck and therefore also distribute the recycling of nodes.

Thus, in comparison with e. g. [16], we extend the concept of garbage collection to include wait-free redistribution. On the other hand, we simplify matters by the assumption that accessible terms are not modified, and by an extension of the repertoire of atomic instructions. Indeed, it is known that consensus needs more than atomic read-write variables and therefore wait-free redistribution requires the strength of consensus variables.

We decided to develop our algorithm on the basis of reference counting cf. [15], because we thought it the most suitable for a wait-free algorithm, for instance because it does not show periods where processors are suspended for garbage collection. It might be interesting, however, but no doubt quite tedious, to come up with correct wait-free algorithms for other forms of garbage collection.

As far as we know, the garbage collection algorithm presented by Herlihy and Moss in [9] was the first lock-free shared-memory multiprocessor algorithm that did not require some form of global synchronization. Our algorithm belongs to the same class. In many respects, however, our algorithm is orthogonal to their proposal. Their memory holds objects of arbitrary size with values that can be modified, but they do not consider trees of objects or deletion of objects. Memory management in [9] therefore consists of the removal of outdated versions. Their approach to a solution is also different: they let each process manage its own portion of shared memory. In contrast to this, all our nodes have the same size, but we can use trees of nodes to store and structure bigger values. In our algorithm, the processes share the management task of the whole memory. This implies that when some processes stop functioning, their management tasks are taken care of by other processes. It is at this point that the algorithm of [9] is not wait-free. Clearly, either algorithm will be blocked when the database is full and no nodes can be recycled. Since the specifications of the algorithms imply that their concepts of recyclability are completely different, it is hard to make a fair comparison between our algorithm and the one of [9].

Algorithms in which concurrent processes manipulate a shared pointer structure are error prone. We therefore provide a proof of the algorithm by means of invariants. Since the verification of invariants when processes concurrently execute array modifications is rather tricky, we have verified the invariants mechanically with the theorem prover Nqthm of Boyer and Moore, cf. [3]. In this paper we give no details of the mechanical aspects of this proof (it is somewhat simpler than the proof in [11]). The mechanical proof is available at the Web site [13].

We did some experiments to test the performance of our algorithm. The experiments indicate a quite satisfactory behaviour, (i. e. linear without any overhead for communication between processes), except when term nodes start travelling between the second level caches of processors on a distributed system.

Overview

In Sect. 2, we describe the data structure and we specify the interface procedures by means of preconditions and postconditions. In Sect. 3, we extend these specifications with safety properties and progress properties. In fact, the interface procedures can be called concurrently by different processes. Therefore, safety properties of the atomic steps are needed. In this section we also prove that the properties imply that the graph remains acyclic.

In Sect. 4 we describe the available repertoire of atomic actions, and we make a start with the construction of the interface procedures. Section 5 deals with aspects of garbage collection in the implementation. In Sect. 6, we construct the remaining interface procedures by combining various procedures constructed before. Section 7 discusses the verification of the properties promised in Sect. 3. In Sect. 8 we describe experiments, which indicate that the algorithm is quite suitable for multiprocessing. Finally, Sect. 9 contains some conclusions.

2 The interface

In this section we describe the memory management interface, as offered to application programmers. It consists of a shared data structure, and a number of procedures that can be used in the application processes. The application programmers are responsible for ensuring that a procedure offered is called only when its precondition holds. It is therefore a proof obligation of the system that the precondition of any interface procedure for a process p is stable under the actions of all processes $\neq p$.

The database is organized as a modifiable directed graph. It is convenient for arguing about correctness to regard the attributes of cells as arrays indexed by the unstructured type *Node*. Therefore, if n is of the type *Node*, the data of n is denoted by $data\,[n]$ (instead of $n.data$ as it would be if n was a record with a field *data*).

So, we have a set *Node* of (numbers of) nodes. We use $0 \notin Node$ to indicate the absence of a node, and define $Node0 = Node \cup \{0\}$. We assume that all nodes n are equivalent, i. e. , have the same maximal degree. We number the children of a node by means of some type *Index*. Therefore, they form a sequence, and the directed graph is given by a variable *children*, according to the declaration

$$\begin{array}{l}\mathbf{type}\ Sequence = \mathbf{array}\ Index\ \mathbf{of}\ Node0;\\ \mathbf{var}\ children : \mathbf{array}\ Node\ \mathbf{of}\ Sequence.\end{array}$$

Thus, $children\,[n]$ is the sequence of children of node n and $children\,[n,i]$ is the ith child of node n.

Each application process maintains a private variable *roots* that holds the nodes the process has direct read access to. We write $roots.p$ for the value of *roots* of process p. We use a predicate $R(p,n)$ to express that process p is allowed to read the data of node n. We shall ensure that predicate $R(p,n)$ can only be invalidated by process p itself. We define

$$R(p,n) \quad \equiv \quad (\exists\, m \in roots.p :: \ m \overset{*}{\to} n),$$

where relation $\overset{*}{\to}$ is the reflexive transitive closure of relation $\to$ on *Node* defined by

$$m \to n \quad \equiv \quad (\exists\, i \in Index :: \ children\,[m,i] = n).$$

For access and modification of the database we provide the application processes with a number of commands, each consisting of a number of atomic instructions. In the presentation the keyword **privar** stands for a private variable of a process (cf. [5]). The following procedure serves to extend the graph with a new node.

```
procedure $Make\ (x : Data,\ y : Sequence,$
                  privar $v : Node)$
{ pre $R^*(self, y) \ \wedge\ roots.self = X$;
  post $roots.self = X \cup \{v\} \ \wedge\ data\,[v] = x$
                       $\wedge\ children\,[v] = y$ } ,
```

where *self* stands for the calling process and X is a specification constant to express the initial value of *roots*. The precondition $R^*(self,y)$ expresses that all children for the new node must be accessible to the caller. Here, accessibility of a sequence y is defined by

$$R^*(p,y) \quad \equiv \quad (\forall\, i \in Index :: \ y\,[i] = 0 \ \ \vee \ \ R(p, y\,[i])).$$

The requirement that procedure *Make* does not change the accessible part of the graph will be expressed in Sect. 3 below.

Under the precondition $R(p, v)$, process p may inspect the contents of node v by calling

procedure *Read* $(v : \mathit{Node},$ **privar** $x : \mathit{Data},$
privar $y : \mathit{Sequence}$)
{ **pre** $R(\mathit{self}, v)$; **post** $x = \mathit{data}\,[v] \ \wedge \ y = \mathit{children}\,[v]$ } .

Access to nodes can be transferred between processes. Assume that $v \notin \mathit{roots}.p$, and that process q satisfies $R(q, v)$ and *has agreed to preserve that* until p has acknowledged reception of node v. Then process p may claim (direct) access to node v by calling

procedure *Accept* $(v : \mathit{Node})$
{ **pre** $v \notin \mathit{roots}.\mathit{self} = X$; **guaranteed** $R(q, v)$;
post $\mathit{roots}.\mathit{self} = X \cup \{v\}$ } .

The application programmer who uses procedure *Accept* has to supply some coordination protocol such that process q does not release node v before process p has completed *Accept*. Note that procedure *Accept* is useful even in the case $q = \mathit{self}$; e. g. when a process wants to remove a tree while retaining a subtree, it can first accept the root of the subtree in its *roots*. In such a case the coordination is trivial.

If it has $v \in \mathit{roots}.p$, process p can relinquish its direct rights on a node v by

procedure *Delete* $(v : \mathit{Node})$
{ **pre** $v \in \mathit{roots}.\mathit{self} = X$; **post** $\mathit{roots}.\mathit{self} = X \setminus \{v\}$ } .

Example. Take $\mathit{Index} = \{1, 2, 3\}$. Let the database be initially empty. Assume process $p0$ executes *Make* $(a, (0, 0, 0), j)$ followed by *Make* $(b, (0, j, 0), k)$. Then $\mathit{roots}.p0 = \{j, k\}$ and j is a child of k. Now $p0$ executes *Delete* (j). Then $p0$ can still access node j via node k. So it can send a message to some process $p1$ with the value of j and $p1$ can execute *Accept* (j) and acknowledge receipt. Then $p0$ may execute *Delete* (k) and we have $\mathit{roots}.p0 = \{\}$ and $\mathit{roots}.p1 = \{j\}$. □

Finally, we provide two procedures for memory management that, at the interface level, are equivalent to *skip* :

procedure *Serve* () ;
procedure *Browse* ().

Procedure *Serve* can be called by an application process that has time to do some garbage collecting. Procedure *Browse* is for a dedicated garbage collecting process. Both procedures are superfluous and only serve for smoother performance. Both are wait-free.

3 System properties

We need safety properties and progress properties. In fact, we want to express that the accessible part of the graph is never modified (safety), and that procedure calls terminate (progress). First some notation to express such properties formally.

We write $p : P \ \triangleright \ Q$ to express that, if precondition P holds and process p performs an atomic action, this action has postcondition Q. We write $P \ \triangleright \ Q$ to express that $q : P \ \triangleright \ Q$ holds for all processes q. We write p **in** *Pd* to express that process p is executing procedure *Pd*. We write $p : P \ o\!\!\rightarrow \ Q$ to express the existence of a constant k such that every execution that starts in a state where P holds and that contains at least k atomic steps of process p, contains a state that satisfies Q.

We characterize the reachable nodes of the graph by

$$\mathit{ER}\ (\mathrm{n}) \quad \equiv \quad (\exists\, q \in \mathit{Process} :: R(q, n)).$$

The main safety properties are that $\mathit{data}\ [n]$ and $\mathit{children}\ [n]$ of a reachable node n are not modified, and that $\mathit{roots}.q$ is modified only when process q itself executes *Make* , *Accept*, or *Delete* . This is formalized in the requirements

(Sq0) $\mathit{ER}\,(n) \ \wedge \ \mathit{data}\ [n] = X \quad \triangleright \quad \mathit{data}\ [n] = X;$
(Sq1) $\mathit{ER}\,(n) \ \wedge \ \mathit{children}\ [n, i] = X$
$\quad \triangleright \quad \mathit{children}\ [n, i] = X;$
(Sq2) $p : \ p \neq q \ \wedge \ \mathit{roots}.q = X \quad \triangleright \quad \mathit{roots}.q = X;$
(Sq3) $p : \ \neg\,(p$ **in** *Delete* $) \ \wedge \ X \in \mathit{roots}.p$
$\quad \triangleright \quad X \in \mathit{roots}.p;$
(Sq4) $p : \ p$ **in** *Delete* $(v) \ \wedge \ v \neq X \ \wedge \ X \in \mathit{roots}.p$
$\quad \triangleright \quad X \in \mathit{roots}.p.$

As before, X is a specification constant (logical variable) to express that the value of a modifiable field is not changed in the step, or that a protected node remains protected.

Wait-free termination of five of the six interface procedures is expressed in

(Sq5) $p : \ p$ **in** *Pd* $\quad o\!\!\rightarrow \quad \neg\,(p$ **in** *Pd* $)$
for *Pd* $\in \{\mathit{Read}, \mathit{Accept}, \mathit{Delete}, \mathit{Serve}, \mathit{Browse}\}.$

Procedure *Make* can only be guaranteed to terminate if there are free nodes to be found. Therefore, in the progress assertion for *Make* , we need an alternative *Full* in the following way:

(Sq6) $p : \ p$ **in** *Make* $\quad o\!\!\rightarrow \quad \neg\,(p$ **in** *Make*$) \ \vee \ \mathit{Full}\,.$

We require that, if *Full* holds, all nodes are in use or there exists a process q that will negate *Full* within a bounded number of steps of q.

Clearly, the alternative *Full* violates wait-freedom, but this is unavoidable, since processes are allowed to claim as much memory as needed. The problem is also slightly complicated by the possibility that a process stops functioning when it is about to make nodes free for reuse. We come back to predicate *Full* in Sect. 7.

We now show that reachability $R(p, n)$ can only be falsified by process p itself, and only in procedure *Delete* . In fact, for processes p, q, and node n, we claim

(Hq0) $p : \ p \neq q \ \wedge \ R(q, n) \quad \triangleright \quad R(q, n);$
(Hq1) $p : \ \neg\,(p$ **in** *Delete*$) \ \wedge \ R(p, n) \quad \triangleright \quad R(p, n).$

Both assertions follow from the definition of $R(q, n)$, via (Sq1), (Sq2), and (Sq3), by induction in the length of the path to node n in the precondition.

It follows from (Sq2) and (Hq0) that, indeed, the precondition of any interface procedure for a process p is stable under the actions of all processes $\neq p$.

We turn to the point that the graph should remain acyclic. For this purpose, we postulate that an unreachable node does not become a new child:

(Sq7) $\neg\,\mathit{ER}\,(n) \ \wedge \ \mathit{children}\ [m, i] \neq n$
$\quad \triangleright \quad \mathit{children}\ [m, i] \neq n.$

It follows from (Sq7) and (Sq1) that we have

(Hq2) $\neg\, ER\,(n) \ \wedge\ \neg\,(m \to n) \quad \rhd \quad \neg\,(m \to n)$,
$ER\,(m) \ \wedge\ \neg\,(m \to n) \quad \rhd \quad \neg\,(m \to n)$.

Now assume that an atomic action has in its postcondition a cycle of nodes $v_i \to v_{i+1}$ for $0 \le i < k$, where $k \ge 1$ and $v_k = v_0$. Then the precondition of this atomic action satisfies, for all i with $0 \le i < k$,

$\neg\, ER\,(v_{i+1}) \ \Rightarrow\ v_i \to v_{i+1}$,
$ER\,(v_i) \ \Rightarrow\ v_i \to v_{i+1}$,
$ER\,(v_i) \ \wedge\ v_i \to v_{i+1} \ \Rightarrow\ ER\,(v_{i+1})$,

by the formulas (Hq2) and the definition of *ER*. It follows that the cycle also existed in the precondition of the atomic action. For, in the precondition, the absence of an edge of the cycle implies that some v_j is reachable, and if some v_j is reachable then all v_j are reachable and all edges are present.

We now assume that, initially, the graph $(Node, \to)$ has no cycles. Then it follows that the graph invariantly has no cycles.

4 The implementation

We turn to a proposal for implementing the system in shared memory.

We use the following repertoire of elementary instructions. Every elementary instruction refers to at most one shared variable, cf. [17], preferably at most once. We have two types of shared variables `t` that can occur more than once in an atomic instruction: counters and consensus variables. Apart from reading and writing, such a variable `t` has one of the special instructions

`t :=  t ± 1` , or `t++ and t--` {counter} ;
if `t = 0` **then** `t :=` w **fi** {consensus} ;

where w is a private variable, and $\pm$ stands for either $+$ or $-$. We assume that modifications of private variables can be combined atomically with an operation on a shared variable. Moreover, we assume that the conditional setting of a consensus variable is combined with the setting of a boolean flag, so that the **then** branch and the virtual **else** branch can be combined with private actions. This is called strong consensus in [11].

In our experiments (see Sect. 8) we had to implement the atomic counter modification by means of a strong compare& swap register, as proposed in [7]:

```
repeat  $tmp := t$ ;
    $\langle\ b := (tmp = t)$ ; if $b$ then $t := tmp \pm 1$ fi $\rangle$
until  $b$.
```

Here and henceforth, the brackets $\langle\ \rangle$ are used to enclose atomic regions. The above loop is not wait-free, but turns out to work satisfactorily.

We now turn to the implementation of the CaRuD interface. It is trivial to implement

```
procedure Read ($v$ : Node, privar $x$ : Data,
                          privar $y$ : Sequence ) =
{ pre  $R(self, v)$; post  $x = data\,[v] \ \wedge\ y = children\,[v]$ }
    $x := data\,[v]$ ;   $y := children\,[v]$
end .
```

We use the notation $v.q$ to refer to the value of a private variable v of process q. For an efficient implementation of *Make* , we give each process a private variable *res* of type **set** of *Node* to hold free nodes reserved for private use. Now one of the problems is to guarantee that, for every process q, if needed, $res.q$ becomes nonempty within bounded delay. Experience seems to show that this must be made a shared responsibility for all processes together. We therefore provide every process q with a consensus variable *waiting* $[q]$ to receive free nodes, according to the declaration

waiting : **array** *Process* **of** *Node0*.

By convention, *waiting* $[p] = 0$ means that process p is waiting for a new node, while *waiting* $[p] = n$ with $n \neq 0$ means that p can use the new node n by means of

```
        procedure receive () =
        { pre  $waiting\,[self] \neq 0$ }
25          $v := waiting\,[self]$;
26          $res := res \cup \{v\}$ ; $waiting\,[self] := 0$
        end .
```

Here each numbered instruction is one atomic command; we give each process q a corresponding instruction pointer $pc.q$. The bigger atomic instruction 26 is allowed since *res* and v are private variables. We use numbered instructions and (below) **goto** s since the concurrency forces us in the invariants to be very precise about where which property holds. Moreover, the use of structured programming with **if** and **while** tends to obscure which instructions are regarded as atomic.

We assume that processes share their wealth in a fair way. For the purpose of redistribution of nodes, we give every process a private variable *fav* of type *Process* (for current favourite). We say that a function *next* traverses a set X (cf. [12]) iff, for every pair $x, y \in X$, there is a number k with $next^k(x) = y$. It follows that $next^k(x) = x$ for $k = \#X$. We give every process a private function *nextp* that traverses *Process*, the set of process numbers, to choose the next favourite. A process may try to share its wealth by executing

```
        procedure share ($v$ : Node ) =
        { pre  $v \in res$ }
29          if  $waiting\,[fav] = 0$  then
                $waiting\,[fav] := v$ ; $res := res \setminus \{v\}$ fi;
30          $fav := nextp\,(fav)$
        end .
```

Here we use the fact that *waiting* is an array of consensus variables and that actions on the private variable *res* may be combined atomically. Note that the value of *res* is retained if the test fails.

For the purpose of garbage collecting, we introduce reference counting by means of a shared array

cnt : **array** *Node* **of** *Integer*

We assume available the atomic increment and decrement operations *cnt* $[n]$ `++` and *cnt* $[n]$ `--`. The idea is that *cnt* $[n]$ estimates the number of edges directed towards n plus the number of processes that have direct access to n. More precisely, we postulate for all nodes n:

(Jq0) $\mathit{cnt}\,[n] = (\#(m,i) \in \mathit{Edge} :: \mathit{children}[m,i] = n)$
$\quad +(\#q \in \mathit{Process} :: n \in \mathit{roots}.q)$
$\quad +(\#q \in \mathit{Process} :: \mathit{waiting}\,[q] = n)$
$\quad +(\#q \in \mathit{Process} :: n \in \mathit{res}.q)$
$\quad +(\#q$ **at** (*) $:: n = w.q)$.

Here *Edge* is the set of pairs (m,i) where m is a node, and i is an index. Since *cnt* $[n]$ is a shared variable which cannot be modified in the same atomic statement as the shared variable *children* $[m,i]$, there are two program locations where *cnt* $[n]$ must be related to a private variable of a process. These locations will be marked with (*). We now use the notation q **at** (*) to indicate that the next action of process q is marked with (*) and we assume that every process q **at** (*) has a private variable w of the type *Node*.

In order to prove that (Jq0) is preserved when a process executes instruction 26 of *receive*, we postulate the invariants

(Jq1) $\mathit{pc}.q = 26 \;\Rightarrow\; \mathit{waiting}\,[q] = v.q$;
(Jq2) $n \in \mathit{res}.q \;\Rightarrow\; \mathit{cnt}\,[n] = 1$.

Note that (Jq0) and (Jq2) together with the typing restriction $\mathit{res}\,.r \subseteq \mathit{Node}$ imply

(Hq3) $\mathit{waiting}\,[q] \notin \mathit{res}\,.r$.

To preserve (Jq1) and (Jq2) in *receive*, we postulate

(Jq3) $24 < \mathit{pc}.q \leq 26 \;\Rightarrow\; \mathit{waiting}\,[q] \neq 0$;
(Jq4) $\mathit{waiting}\,[q] \neq 0 \;\Rightarrow\; \mathit{cnt}\,[\mathit{waiting}\,[q]] = 1$.

At this point the reader is invited to verify that (Jq0), (Jq1), (Jq2), (Jq3), (Jq4) are preserved by all atomic actions in the procedures *Read*, *receive*, and *share*. Note that the preconditions of *receive* and *share* are used in these verifications.

We can now easily implement *Accept*.

```
     procedure Accept $(v : \mathit{Node}) =$
     { pre $v \notin \mathit{roots.self} = X$; guaranteed $R(q,v)$;
       post $\mathit{roots.self} = X \cup \{v\}$ }
33     $\mathit{cnt}\,[v]$ ++ ;  $\mathit{roots} := \mathit{roots} \cup \{v\}$
     end .
```

To prove that *Accept* preserves (Jq2) and (Jq4), we use the guarantee $R(q,v)$ together with the predicates

(Hq4) $n \in \mathit{res}.q \;\Rightarrow\; \neg\,\mathit{ER}\,(n)$,
(Hq5) $n = \mathit{waiting}\,[q] \;\Rightarrow\; \neg\,\mathit{ER}\,(n)$,

which follow from (Jq0) and (Jq2).

We introduce a shared variable *clean* of type **array** *Node* **of** *Boolean* with the invariant

(Jq5) $\mathit{clean}\,[n] \;\Rightarrow\; \mathit{children}\,[n,j] = 0$.

We now turn to the implementation of *Make* . The top level design is as follows.

```
procedure Make $(x : \mathit{Data},\ y : \mathit{Sequence},$ privar $v : \mathit{Node}) =$
{ pre $R^*(\mathit{self},y) \;\wedge\; \mathit{roots.self} = X$;
  post $\mathit{roots.self} = X \cup \{v\} \;\wedge\; \mathit{data}\,[v] = x$
                 $\wedge\; \mathit{children}\,[v] = y$ }
   get () ;
   choose $v \in \mathit{res}$ ;
   branch $(x,y,v)$
end .
```

Here, procedure *get* must enable the choice in *res* by making *res* nonempty; its implementation is postponed till Sect. 6. Procedure *branch* inserts node v into the graph with data x and sequence of children y. Its conventional code is

```
procedure branch $(x : \mathit{Data},\ y : \mathit{Sequence}, v : \mathit{Node}) =$
{ pre $R^*(\mathit{self},y) \;\wedge\; \mathit{roots.self} = X \;\wedge\; v \in \mathit{res.self}$
                $\wedge\; \mathit{clean}\,[v]$ ;
  post $\mathit{roots.self} = X \cup \{v\} \;\wedge\; \mathit{data}\,[v]$
                $= x \;\wedge\; \mathit{children}\,[v] = y$ }
   $\mathit{data}\,[v] := x$ ;
   $\mathit{clean}\,[v] := \mathit{false}$ ;
   for all $i \in \mathit{Index}$ do
      if $y[i] \neq 0$ then
         $\mathit{cnt}\,[y[i]]$ ++ ;
         $\mathit{children}\,[v,i] := y[i]$ fi
   od ;
   $\mathit{roots} := \mathit{roots} \cup \{v\}$ ;  $\mathit{res} := \mathit{res} \setminus \{v\}$ ;
end .
```

Since we want a fine grain of atomicity for all instructions concerning shared memory, we need invariants concerning the state at all locations in the loop. This forces us to make the jumps explicit and to introduce a private variable F for the set of indices that have not yet been treated in the loop. We thus encode the loop over the indices by means of three jumps and a shrinking set of indices F. Omitting the specification, we get

```
      procedure branch $(x : \mathit{Data},\ y : \mathit{Sequence},$
      $v : \mathit{Node}) =$
41       $\mathit{data}\,[v] := x$;
42       $\mathit{clean}\,[v] := \mathit{false}$ ;  $F := \mathit{Index}$;
43       if $F = \emptyset$ then goto 47 fi ;
44          choose $i \in F$ ;  $F := F \setminus \{i\}$;
            $w := y\,[i]$ ;  if $w = 0$ then goto 43 fi ;
45          $\mathit{cnt}\,[w]$ ++;
46    (*)   $\mathit{children}\,[v,i] := w$ ;  goto 43;
47       $\mathit{roots} := \mathit{roots} \cup \{v\}$ ;  $\mathit{res} := \mathit{res} \setminus \{v\}$;
48    end .
```

Note that instruction 46 has the star (*). Indeed, the private variable w is introduced here for the sake of (Jq0).

To prove preservation of (Jq0) in 46, of (Jq2) at 45, and of (Jq5) at 46, we postulate the invariants

(Jq6) $44 < \mathit{pc}.q \leq 46 \;\Rightarrow\; \mathit{children}\,[v.q, i.q] = 0$;
(Jq7) $40 < \mathit{pc}.q \leq 47 \;\wedge\; y.q\,[j] \neq 0 \;\Rightarrow\; R(q, y.q\,[j])$;
(Jq8) $42 < \mathit{pc}.q \leq 47 \;\Rightarrow\; \neg \mathit{clean}\,[v.q]$.

For preservation of (Jq6) in 44, we postulate

(Jq9) $42 < \mathit{pc}.q \leq 46 \;\wedge\; j \in F.q$
$\quad \Rightarrow\; \mathit{children}\,[v.q, j] = 0$.

Preservation of (Jq6) when another process executes 46 will follow from (Jq0) and (Jq2).

Preservation of (Jq9) in 42 follows from (Jq5) and the new postulate

(Jq10) $v.q \in \mathit{res}.q \;\wedge\; \neg\,\mathit{clean}\,[v.q] \;\Rightarrow\; 42 < \mathit{pc}.q \leq 47$.

Here it must be mentioned that, in the invariants and in the mechanical proof, we treat the parameters and the local variables of the procedures as persistent private variables of the processes. With regard to the invariants, these variables are allowed to be modified nondeterministically before every procedure call, only subject to the precondition of the call. In particular, for process q, all parameters v of procedures called by q are modelled by the same private variable $v.q$. This unusual treatment of local variables simplifies the proofs of the invariants and even their interpretation.

Preservation of (Jq10) is proved by means of the new postulates

(Jq11) $\mathit{waiting}\,[q] \neq 0 \;\Rightarrow\; \mathit{clean}\,[\mathit{waiting}\,[q]]$;
(Jq12) $n \in \mathit{res}.q \;\wedge\; \neg\,\mathit{clean}\,[n] \;\Rightarrow\; n = v.q$.

At this point the above invariants (Jq. . .) can all be proved. In these proofs we also use the following obvious invariants, which are only concerned with the private variables of a single process:

(Pq0) $\mathit{pc}.q = 29 \;\Rightarrow\; v.q \in \mathit{res}.q$;
(Pq1) $\mathit{pc}.q = 33 \;\Rightarrow\; v.q \notin \mathit{roots}.q$;
(Pq2) $40 < \mathit{pc}.q \leq 47 \;\Rightarrow\; v.q \in \mathit{res}.q$;
(Pq3) $44 < \mathit{pc}.q \leq 46 \;\Rightarrow\; w.q = y.q\,[i.q] \neq 0$;
(Pq4) $44 < \mathit{pc}.q \leq 46 \;\Rightarrow\; i.q \notin F.q$.

Of course, we also need the application guarantee of *Accept* :

(AG) $\mathit{pc}.q = 33 \;\Rightarrow\; \mathit{ER}\,(v.q)$.

Finally, for the specification of *branch*, we observe that $\mathit{pc}.q = 48$ implies $\mathit{children}\,[v.q, j] = y.q\,[j]$, as follows from the invariants

(Jq13) $42 < \mathit{pc}.q \leq 48$
$\quad\Rightarrow\; \mathit{children}\,[v.q, j] = y.q\,[j] \;\vee\; j \in F.q$
$\quad\quad\vee\; (j = i.q \wedge 44 < \mathit{pc}.q \leq 46)$;
(Pq5) $46 < \mathit{pc}.q \leq 48 \;\Rightarrow\; F.q = \emptyset$.

Note that proofs of invariance may be circular: the assumption is that all invariants hold in the precondition of every atomic instruction, and for each invariant one then proves that it holds in the postcondition of every instruction.

5 Garbage collection

We now come to the point where nodes are made free again. If $n \in \mathit{roots}.p$ holds, process p may relinquish its rights on n (and the dependent nodes) by removing n from $\mathit{roots}.p$ and decrementing $\mathit{cnt}\,[n]$.

It is attractive to combine this with garbage collection, if $\mathit{cnt}\,[n] = 1$ holds in the precondition. This idea is not sufficient for garbage collection since $\mathit{cnt}\,[n] > 1$ is not stable in the precondition: two processes may observe that $\mathit{cnt}\,[n] = 2$ and both decide to decrement $\mathit{cnt}\,[n]$ without garbage collection. We therefore decide to do garbage collection for nodes with $\mathit{cnt}\,[n] = 0$. We thus implement procedure *Delete* as follows.

```
      procedure Delete $(v : \mathit{Node})$
      { pre $v \in \mathit{roots.self} = X$;
        post $\mathit{roots.self} = X \setminus \{v\}$ }
65      $\mathit{cnt}\,[v]$ -- ;   $\mathit{roots} := \mathit{roots} \setminus \{v\}$;
66      collect $(v)$
      end .
```

To prove preservation of (Jq0), we use the invariant

(Pq6) $\mathit{pc}.q = 65 \;\Rightarrow\; v.q \in \mathit{roots}.q$.

In procedure *collect* the decrementing process tests whether decrementing $\mathit{cnt}\,[n]$ established $\mathit{cnt}\,[n] = 0$ and then adds n to a private variable *list* which holds a bounded list of nodes n that are likely to satisfy $\mathit{cnt}\,[n] = 0$. The variable *list* is important for the performance of the system, but is formally superfluous: it does not occur in the invariants.

```
      procedure collect $(v : \mathit{Node}) =$
         if $\mathit{cnt}\,[v] = 0$
         then $\mathit{list} := \mathit{truncate}\;(v : \mathit{list})$ fi
      end .
```

Here v is placed at the head of the list *list* and, if in this way *list* becomes too long, the last element of *list* is removed. Note that the elements $v \in \mathit{list}.p$ need not satisfy $\mathit{cnt}\,[v] = 0$ since other processes may have incremented $\mathit{cnt}\,[v]$, by reclaiming v via the node distribution strategy described in Sect. 6.

It follows from (Jq0) that $\mathit{cnt}\,[n] = 0$ implies that node n is free. A free node can be claimed by any process that needs new nodes. Therefore, if two or more processes want to claim the same free node they need consensus to decide which claimant succeeds. We therefore introduce locking of nodes, by means of shared consensus variables

lock : **array** *Node* **of** *Boolean*.

We now introduce the garbage collecting procedure *untarget* that tries to obtain a node v for *res*, after resetting its targets if necessary.

```
      procedure untarget $(v : \mathit{Node}) =$
51       if $\neg\mathit{lock}\,[v]$ then $\mathit{lock}\,[v] := \mathit{true}$
            else return fi ;
52       if $\mathit{cnt}\,[v] \neq 0$ then goto 61 fi;
53       if $\mathit{clean}\,[v]$ then goto 60 else $F := \mathit{Index}$ fi;
54       if $F = \emptyset$ then goto 59 fi ;
55          choose $i \in F$ ;   $F := F \setminus \{i\}$;
            $w := \mathit{children}\,[v, i]$ ;
            if $w = 0$ then goto 54 fi ;
56          $\mathit{children}\,[v, i] := 0$ ;
57    (*)   $\mathit{cnt}\,[w]$ -- ;
58          collect $(w)$ ;   goto 54;
59       $\mathit{clean}\,[v] := \mathit{true}$;
60       $\mathit{cnt}\,[v] := 1$ ; $\mathit{res} := \mathit{res} \cup \{v\}$;
61       $\mathit{lock}\,[v] := \mathit{false}$
      end .
```

Note that *lock* is an array of “strong” consensus variables; see the atomic instruction 51.

To prove preservation of (Jq0) in 56, 57, 60, we postulate

(Kq0) $\mathit{pc}.q = 56 \;\Rightarrow\; w.q = \mathit{children}\,[v.q, i.q]$;
(Kq1) $52 < \mathit{pc}.q \leq 60 \;\Rightarrow\; \mathit{cnt}\,[v.q] = 0$.

Preservation of (Jq5) follows from

(Kq2) $\mathit{pc}.q = 59 \;\Rightarrow\; \mathit{children}\,[v.q, j] = 0$.

Preservation of (Jq10) at 60 follows from

(Kq3) $\mathit{pc}.q = 60 \;\Rightarrow\; \mathit{clean}\,[v.q]$.

Preservation of (Kq1) under *Accept* and instruction 45 follows from

(Hq6) $\mathit{cnt}\,[n] = 0 \;\Rightarrow\; \neg\,\mathit{ER}\,(n)$, which follows from (Jq0). Preservation of (Kq2) in 54 follows from

(Kq4) $53 < pc.q \leq 58 \ \wedge \ children\,[v.q, j] \neq 0$
$\wedge \ j \notin F.q$
$\Rightarrow \ j = i.q \ \wedge \ pc.q = 56.$

We now have to prove that the above invariants, especially (Kq0) and (Kq1), are not violated by another process that executes *untarget*. So we want to have interference freedom, as expressed by

(Kq5) $51 < pc.q \leq 61 \ \wedge \ 51 < pc.r \leq 61$
$\wedge \ v.q = v.r \ \Rightarrow \ q = r.$

This is accomplished by locking. Preservation of (Kq5) easily follows from the invariant

(Kq6) $51 < pc.q \leq 61 \ \Rightarrow \ lock\,[v.q].$

Preservation of (Kq6) is proved by means of (Kq5).

6 A strategy for redistribution

The elements of *list* are good candidates for procedure *untarget*. If *list* is empty, however, the process can choose an arbitrary node, if it does so in a fair way. We therefore give every process a private variable *nod* and a private function *nextn* that traverses the set *Node*.

We now present procedure *get* that was used in *Make* to make the set *res* nonempty.

```
procedure get () =
    while res = ∅ do
        if waiting [self] ≠ 0
         then receive () else search () fi
     od
end .
```

where

```
procedure search () =
    if list = ∅ then v := nod ; nod := nextn (nod)
    else v := head (list) ; list := tail (list) fi ;
    untarget (v) ;
    if v ∈ res then share (v) fi ;
end .
```

The call of *share* in *search* is needed to guarantee wait-freedom: each process is served within bounded delay (although the bound depends on the number of processes and the size of the memory).

In Sect. 7, we'll show that procedure *get* is wait-free provided there are always enough free nodes to be found.

We turn to memory management activities that are invisible at the interface level. The application processes are allowed to accumulate nodes in their sets *res*, provided they try to share and *res* does not become too large. We therefore provide each process with a private constant *maxres* ≥ 1, with the invariant

(Pq7) $\#res.q \ \leq \ maxres.q.$

It is easy to see that (Pq7) is only threatened by *untarget*, and that it is preserved when *untarget* is called with precondition $\#res.p < maxres.p.$

Therefore, whenever process q has time to do some garbage collecting, it may call

```
procedure Serve () =
    if #res < maxres then search ()
    else choose v ∈ res ; share (v) fi
end .
```

For the sake of efficiency it may be preferable to have one additional garbage collecting process *gc* with *maxres* $.gc = 1$. Process *gc* only frequently calls

```
procedure Browse () =
    search () ;
    if res ≠ ∅ then
        choose v ∈ res ;
        ⟨ res := ∅ ; cnt [v] := 0 ⟩
     fi
end .
```

Procedure *Browse* detects free nodes with sons. It frees the sons, by decreasing their incounters, thus making it easier for the other processes to find free nodes. It preserves (Pq7) since it has *res* $.gc = \emptyset$ in the idle states.

Remark. The conditional jump in 53 of *untarget* is only useful if *Index* is large and the probability of $cnt\,[v] = 0 \wedge clean\,[v]$ is sufficiently high. In particular, the jump is useless if we have the invariant

$clean\,[n] \ \Rightarrow \ (\exists\, q :: n = waiting\,[q] \ \vee \ n \in res.q$
$\vee \ (n = v.q \ \wedge \ pc.q = 60)).$

This predicate is preserved by all procedures except for *Browse*. So, in a system that does not use *Browse* or in which *Index* is very small, we had better remove the jump and replace instruction 53 by

53' $F := Index;$

In that case, variable *clean* becomes a ghost variable and can therefore be removed from the algorithm. □

7 Verification of properties

It remains to verify the global properties (Sq0) through (Sq7). Since *data* and *children* are modified only in *branch* and *untarget*, the properties (Sq0), (Sq1), (Sq7) follow from (Hq4) and (Pq2), and (Hq6) and (Kq1), and the specification of *branch*. Since *roots* is a private variable, the validity of (Sq2), (Sq3), and (Sq4) is easily verified. The loops in *branch* and *untarget* are bounded by the size of *Index*. Therefore, the only unbounded loop occurs in *get*. Since *get* is only used in the interface procedure *Make* , this implies that the other interface procedures are wait-free, i. e. , (Sq5).

In order to prove (Sq6), we define predicate *Full* by

$Full \ \equiv \ (\forall\, n :: cnt\,[n] > 0 \ \vee \ lock\,[n]).$

Now, informally, property (Sq6) is shown as follows. If *Full* is false during an execution sequence in which process p executes *get*, there are always unlocked nodes n with $cnt\,[n] = 0$. After having exhausted its list *list*, process p traverses the set *Node* and eventually finds unlocked nodes with $cnt\,[n] = 0$. It executes *untarget* on every such node, and then calls *share*; this advances $fav.p$. Therefore, if process p does not terminate early enough, a state is reached with $fav.p = p$. Then process p serves itself, and the call of *get* terminates. The argument can be made formal by means of the techniques developed in [12].

We finally show that, if *Full* holds, then all nodes are in use, or there is a process that will negate *Full* within a bounded number of steps. For this purpose, we formalize “node n being in use” by predicate $\mathit{Used}\,(n)$ defined by

$$\mathit{Used}\,(n) \equiv \mathit{ER}\,(n) \quad \vee \quad (\exists\, q :: \mathit{waiting}\,[q] = n \vee n \in \mathit{res}.q).$$

The definition of *Used* together with the invariants (Jq5), (Jq6), (Jq9), (Jq10), (Jq11) implies

(Hq7) $\mathit{Used}\,(m) \ \wedge\ m \to n \ \Rightarrow\ \mathit{Used}\,(n).$

We now define

$$\begin{aligned} \mathit{LL}\,(n) \quad \equiv \quad & (\mathit{cnt}\ [n] = 0 \ \wedge\ \mathit{lock}\ [n]) \\ & \vee\ \ (\exists\, q :: \mathit{pc}.q = 57 \ \wedge\ n = w.q). \end{aligned}$$

Let a node n be called an *orphan* iff it does not have a parent, i. e. , iff $\neg\,(m \to n)$ holds for every node m.

Lemma. *Let node n be an orphan with $\neg\,\mathit{Used}\,(n)$. Then we have*

(a) $\mathit{cnt}\ [n] \ = \ (\#q ::\ \mathit{pc}.q = 57 \ \wedge\ n = w.q);$
(b) $\mathit{Full} \ \Rightarrow\ \mathit{LL}\,(n).$

Proof. (a) Since n is an orphan, it follows from (Jq0) and $\neg\,\mathit{Used}\,(n)$ that $\mathit{cnt}\ [n] \ = \ (\#q\ \textbf{at}\ (*) :: n = w.q)$. The marker (*) only occurs at 46 in *branch*, and at 57 in *untarget*. The assertion follows, since $w.q$ is *Used* at 46 because of (Pq3).

(b) If $\mathit{cnt}\ [n] > 0$, then $\mathit{LL}\,(n)$ follows from (a). If $\mathit{cnt}\ [n] = 0$, then $\mathit{LL}\,(n)$ follows from *Full*. □

Since the graph $(\mathit{Node}, \to)$ is acyclic, every node has an ancestor that is an orphan. If a node n is not *Used*, every ancestor of n is also not *Used* because of (Hq7). Therefore, the Lemma implies

Theorem. *Assume that Full holds. Then every node n satisfies*

$$\mathit{Used}\,(n) \ \vee\ (\exists\, m :: \mathit{LL}\,(m) \ \wedge\ m \overset{*}{\to} n). \quad \square$$

This theorem shows the absence of memory leakage. In fact, for every unreachable node n with $\mathit{LL}\,(n)$, there is a unique process q that will release n within a bounded number of steps. Therefore, if *Full* holds and not all nodes are *Used*, there exists at least one process that will negate *Full* within a bounded number of steps. Note that establishing $\neg\,\mathit{Full}$ is not wait-free, since the actions of a specific process may be required.

The efficiency of redistribution is hard to estimate. If every process claims new nodes more or less in the same rate as it relinquishes old ones, communication of the nodes plays no significant role. If these rates differ wildly, however, the sets *list* of some processes are often empty, in which case these processes to some extent rely on the charity of other processes in procedure *share*, although they also get new nodes by inspection of arbitrary nodes. Then it is important that congestion of node inspections is avoided by making the traversal functions *nextn* of the processes all different, see [10], Sect. 8.1. Similarly, to avoid congestion of charity, one should take the traversal functions *nextp* all different.

Table 1. Making terms in million nodes/sec on a SGI Origin 2000

sharing:	none	1/10	full	none	1/10	full
$\#\mathit{list}$:	0	0	0	12	12	12
$\#P = 1$	0.59	0.46	0.53	1.2	1.1	0.95
2	0.15	0.12	0.15	1.9	1.5	0.83
4	0.13	0.15	0.13	4.0	2.1	0.70
8	0.20	0.15	0.13	7.9	3.3	0.85
16	0.17	0.15	0.14	11.5	4.4	0.81

8 Performance

How does the algorithm that we present perform? From a theoretical viewpoint the answer is easy. All operations except *Make* operate in constant time. Assuming that there are always sufficiently many free nodes available, *Make* also operates in constant time on average.

This does not answer the question, however, whether our algorithm is competitive from a practical viewpoint. In particular, the compare and swap operator may turn out to be a bottleneck. Also, some architectures of parallel systems may be unfavourable to the algorithm. They may cause substantial constant overhead to the operations and thus may make more classical solutions to parallel memory management preferable.

We implemented the algorithm and ran it on a single processor SGI O2 (180 Mhz) and on a multiprocessor SGI Origin 2000 (with 250Mhz and 300Mhz processors), for different numbers of processes, to get an impression whether the algorithm scales up linearly, as was expected. We let each process iteratively construct, inspect and delete binary trees with 63 nodes. We studied the effect of *list* by taking as its lengths 0 and 12. It turned out that, on the SGI Origin, performance is quite negatively influenced by sharing. Therefore we obtained benchmarks where sharing is on, where sharing only takes place once in every 10 times (but in this case the algorithm is still wait-free), and where sharing of nodes is switched off completely.

Our conclusion is that *reading* nodes scales up linearly with the number of processors (where we managed to read 10^8 nodes per second using 16 processors).

Making nodes is heavily influenced by the length of *list*. On the single processor machine, a list length of 0 forces the processes to search for the nodes, whereas a list length of 12 takes care that free term nodes can be picked up immediately. The former was approximately 3 times slower than the latter ($2 \cdot 10^5$ versus $6 \cdot 10^5$ makes per second). Sharing nodes leads to a 25% speed increase when the list has length 12, compared to not or sometimes sharing. The number of processes does not have any effect on the speed of the system meaning that our algorithm scales up perfectly. For comparison we implemented a straightforward P and V synchronisation solution to the CaRuD problem, which showed a substantial performance degradation when the number of processes are increased.

From this we conclude that one of the strongest reasons to study wait-free or lock-free algorithms is their almost perfect scalability.

Making nodes on the parallel machine gives a much more diverse picture. We provide the data that we obtained in Table

1. The first column gives the number of processors used. The other entries contain the number (in millions) of terms that can be made per second. When *list* has length 0, it appears that adding processors does not lead to any speed increase. The reason for this is that term nodes reside in second level cache, and travel from processor to processor. This completely dominates performance.

Sharing of nodes on the parallel machine also leads to reduced performance. Only non sharing, and a sufficiently long *list* leads to sufficient locality of termnodes to observe a linear speed up.

Note that even in those cases where adding processors does not appear beneficial for making nodes, the algorithm can still be very fruitfully applied on parallel machines, since there can be a substantial speed up in reading and manipulating the nodes. However, further adaption of the algorithm to maximize locality of nodes may lead to an even better performance.

9 Conclusions

The CaRuD interface is a useful and viable abstraction for a graph-like data structure shared by concurrent processes. We show this using a wait-free algorithm that scales up perfectly in theory, and quite well in practice, showing that the wait-free paradigm is a good technical means to obtain high performance on parallel computers. Our algorithm can be improved for cache based multiprocessor machines, such as the Origin 2000, by forcing a better locality to term nodes. We leave this, however, for further research.

From the experience we obtained writing this article we believe that it would be useful to develop wait-free versions of standard sequential algorithms to obtain efficient, fault tolerant parallel versions. One may think of wait-free mark and sweep garbage collectors, to circumvent the acyclicity constraint of reference counting garbage collectors, of wait-free equivalents for linked lists, to circumvent searching the array of term nodes (see [18] for a lock-free implementation), and of wait-free hashtables, to efficiently implement sharing of terms [6].

We found, however, that it is much more difficult to design these algorithms, especially on the level of the ordering of individual assignments. It is not without reason that we resorted to the Boyer-Moore theorem checker for verification of the invariants that we needed. We also would strongly favour the situation where such checkers would become a common tool for the designers of parallel algorithms.

Acknowledgements. We are grateful for comments and suggestions of Jan Jongejan, Pieter Olivier, John Tromp, Robert van Liere, Stefan Manegold, and three anonymous referees.

References

1. Bergstra, J. A. , Klint, P. : The discrete time ToolBus – a software coordination architecture. Sci Comput Program **31**: 205–229 (1998)
2. Brand, M. G. J. van den, Jong, H. A. de, Klint, P. , Olivier, P. A. : Efficient annotated terms. Softw, Pract Exper **30**: 259–291 (2000)
3. Boyer, R. S. , Moore, J. S. : A Computational Logic Handbook. Boston: Academic Press 1988
4. Buhrman, H. , Garay, J. A. , Hoepman, J. H. , Moir, M. : Long-lived renaming made fast. In: *14th Ann. Symp. on Principles of Distributed Computing* (Ottawa, Ont. , Canada) ACM Press 1995, pp. 194–203
5. Dijkstra, E. W. : A discipline of programming. Englewood Cliffs, NJ: Prentice–Hall 1976
6. Groote, J. F. , Hesselink, W. H. : An efficient lock-free algorithm for parallel accessible hashtables. In preparation
7. Herlihy, M. P. : A methodoly for implementing highly concurrent data structures. Proc. 2nd ACM symp. on Principles & Practice of Parr. Progr. , 1990, 197–206
8. Herlihy, M. P. : Wait–free synchronization. ACM Trans. on Program. Languages and Systems **13**: 124–149 (1991)
9. Herlihy, M. P. , Moss, J. E. B. : Lock-free garbage collection for multiprocessors. IEEE Trans Parall Distrib Syst **3**: 304–311 (1992)
10. Hesselink, W. H. : Bounded Delay for a Free Address. Acta Informatica **33**: 233–254 (1996)
11. Hesselink, W. H. : The design of a linearization of a concurrent data object. In: D. Gries, W. -P. de Roever (eds.): Programming Concepts and Methods, Proceedings Procomet ’98, Chapman & Hall, IFIP 1998, pp. 205–224
12. Hesselink, W. H. : Progress under bounded fairness. Distrib Comput 12: 197–207 (1999)
13. Hesselink, W. H. : http://www. cs. rug. nl/˜wim/crud
14. Hesselink, W. H. , Groote, J. F. : Waitfree distributed memory management by Create, and Read Until Deletion (CRUD). Report SEN-R9811, CWI, Amsterdam 1998
15. Jones, R. , Lins, R. : Garbage Collection, algorithms for automatic dynamic memory management. New York: Wiley 1996
16. Jonker, J. E. : On-the-fly garbage collection for several mutators. Distrib Comput **5**: 187–199 (1992)
17. Owicki, S. , Gries, D. : An axiomatic proof technique for parallel programs. Acta Informatica **6**: 319–340 (1976)
18. Valois, J. D. : Lock-free linked lists using compare-and-swap. *Proceedings of the 14th Annual Principles of Distrib Comput*, pages 214–222, 1995. See also J. D. Valois. ERRATA. Lock-free linked lists using compare-and-swap. Unpublished manuscript, 1995

Wim H. Hesselink received his PhD in mathematics from the University of Utrecht in 1975. He then moved to the University of Groningen to work in the field of algebraic groups and Lie algebras. Around 1983, after the loss of a newborn child, he left pure mathematics and found a new challenge in computer science. In 1986/1987 he was on sabbatical leave with E. W. Dijkstra at the University of Texas at Austin. He wrote a book on the weakest preconditions of recursive procedures. Since 1994, he holds a chair for Program Correctness at the Department of Computing Science at the University of Groningen. His current research concentrates on the design of algorithms, if necessary with verification by means of a mechanical theorem prover.

Jan Friso Groote obtained an engineering degree in computer science in 1988 at Twente University in the Netherlands. He received his PhD degree at the University of Amsterdam in 1991 for work done at the Centre for Mathematics and Computer Science (CWI) in Amsterdam on operational semantics and process algebra. Currently he is a full professor at Eindhoven University of Technology in the Netherlands. His main interest lies in improving the state of the art in program and algorithmic design by making the use of mathematical techniques in this process feasible.