The design of a linearization of a concurrent data object

W.H. Hesselink
Dept. of Mathematics and Computing Science, University of Groningen
Postbox 800, 9700 AV Groningen, The Netherlands
Web site: `http://www.cs.rug.nl/~wim`
email: `wim@cs.rug.nl`, *fax +31.50.3633800*

Abstract

A design is presented for fault tolerant linearization of a concurrent data object in shared memory. Server processes may stop and restart functioning without notification, but always at least one member of each family of server processes makes progress. Then every client process that makes progress gets its transactions done within bounded delay.

Invariants are used to prove the safety properties. The processes are tightly coupled but the separation into client processes and some families of server processes enables modularity in the design. Preservation of the invariants is proved by means of the theorem prover Nqthm of Boyer and Moore.

Keywords

Linearization, shared memory, concurrent data object, fault tolerance, memory management, client server architecture, theorem proving

1 INTRODUCTION

A concurrent data object is a data structure shared by concurrent processes. It has a number of client processes that send invocations to the data object and receive corresponding responses. Logically, the object behaves as if the invocations are processed in some sequential order. This requirement is formalized in the concept of linearizability, see (Herlihy *et al.* 1990). Linearizability can be achieved by temporarily blocking some processes, but this has the disadvantage that, if a blocking process is delayed or stopped, other processes are delayed or stopped as well.

In a waitfree data object, cf. (Herlihy 1991), every process completes its invocation in a bounded number of atomic actions, regardless of the actions and execution speeds of the other processes. A waitfree data object is fault tolerant in the sense that, if some process stops executing, the invocations of other processes are not affected. For an object to be waitfree, every client must be able to treat its own invocations, since all other processes are allowed to fail. Waitfreedom is thus a clean but rigid concept. In particular, it does not allow us to delegate subtasks to separate processes. With the aim of fault tolerance and separation of concerns, it is therefore better to require progress under somewhat stronger assumptions on the distribution of failures.

We use the term bounded delay to express the assumption that *the tasks of the system are distributed over a number of families of related processes*, and the requirement that *progress is guaranteed if, and only if, each of these families contains at least one active process*. There is no requirement that an inactive process remains inactive, but if it becomes active again, it restarts at the point where it stopped and with all its previous information.

The stronger fault model allows us to delegate subtasks to separate processes. The resulting separation of concerns leads to much greater elegance and ease of design. Of course, the delegation destroys waitfreedom but the resulting design is much more flexible. It can be tuned to the requirements of the specific situation. Finally, the design can easily be converted into a waitfree implementation: enlarge every client process to a package that also contains one member of each family of server processes, and then schedule the processes in the package in a round robin fashion.

Inspired by (Herlihy 1991), we presented in (Hesselink 1995) a waitfree implementation of an arbitrary concurrent data object in shared memory. Here we present a related design of a linearization with bounded delay of such a data object. We eliminate the assumptions of safe registers and determinacy of the object. We make slightly stronger assumptions on the consensus registers (see Section 2), in order to improve the space and time complexity.

Overview of the paper

In Section 2 we present the general setting: the shared variable model with interleaving semantics, the atomic instructions used, the concepts of invariants and bounded delay, and the aspects of theorem prover assistance. Section 3 contains the formalization of the safety requirements. In Section 4 we present a version of the design without memory management and without proofs.

We then enter the central stage of the design with the verification that the informally motivated design up to this point meets its specification. Indeed, in the Sections 5, 6, 7, we prove centralized safety of this design, i.e., preservation of the first main invariant under the three families of processes Client, Linearizer, and Applier. Section 8 contains the proof of the second main invariant, which expresses that the client processes are served correctly. Section 9 contains the proof of the regularity requirements for read–write variables of larger values.

The Sections 5 up to 8 are enough to indicate our ways to find and preserve invariants. An important issue of the design, however, is that it should work in bounded memory. We therefore have to implement memory management.

Here we use the same methods as above, but, since we do not want to stretch the readers' patience unduly, we start skipping most of the details.

In Section 10, we extend the processes Client, Linearizer, Applier in such a way that efficient distributed garbage collection is possible. We then give the two garbage collecting programs Collector and Distributor for the spaces of addresses and locations. Section 11 contains the initialization. Section 12 contains concluding remarks.

We present around fifty invariants for the design without memory management and skip the eighty invariants for memory management. The reader is not expected to verify that the invariants presented are indeed preserved. We have used a theorem prover for that purpose. Rather, the reader may concentrate on how the invariants are found, and whether they are conceivable and serve some purpose.

2 THE MODEL AND THE REPERTOIRE

Concurrent data objects are reactive systems: they maintain an ongoing interaction with the environment, which is represented by the client processes.

The model of concurrency is as follows. There is a set of processes that communicate by means of shared variables. Every process also has private variables. The (global) state of the system consists of the values of all variables, the instruction pointers included. A step of the system is a transition between states in which one process executes its current instruction. An execution sequence is a sequence of steps. A state is called reachable iff it occurs in an execution sequence that starts in an initial state.

It follows that a single instruction is always treated as atomic. Certain instructions, like read or write instructions of large values, however, are regarded as taking time. This point is formalized as follows. If the current instruction of one process is to write a large value at address v, it is forbidden that the current instruction of another process is to read or write at address v. This is an additional proof obligation. We come back to it in Section 9.

It has been shown in (Herlihy 1991) that atomic read–write registers are not sufficient to construct arbitrary data objects, but that such objects can be constructed if one also allows so–called consensus registers, cf. (Fischer *et al.* 1985, Herlihy 1991).

A consensus register is a shared variable, say $\mathtt{x}$, with atomic actions $u := \mathtt{x}$ and $\mathtt{x} := 0$, and an atomic setting actions

$$\langle\ \mathbf{if}\ \mathtt{x} = 0\ \ \mathbf{then}\ \ \mathtt{x} := u\ \ \mathbf{fi}\ \rangle$$

where 0 is some constant and u is a private variable.

For the sake of efficiency, we also use a somewhat stronger version of consensus. A shared variable $\mathtt{x}$ is called a strong consensus object if the atomic setting action also sets a "status bit", say b, to indicate whether the setting action has been performed. Since atomic operations can always be combined with actions on private variables, we can replace the action on b by other private actions, see the instructions 60, 77, 91, 95 in Section 10.

We also use compare and swap registers. A shared variable $\mathtt{x}$ is called a compare and swap register if it allows instructions of the form

$$\langle \textbf{ if } \mathtt{x} = u \textbf{ then } \mathtt{x} := v \textbf{ fi } \rangle$$

where u and v are private variables of the same type as $\mathtt{x}$.

For the purpose of efficient garbage collection, we also use counter variables, which allow an atomic test for equality with zero and can be atomically incremented and decremented.

Certain variables do not belong to the algorithm proper, but only serve in the proof (and possibly the specification) of the algorithm. Such variables are called ghost variables, or auxiliary variables (Owicki *et al.* 1976) (3.6), or also history variables. The values of these variables may not influence the flow of control or values of the proper variables. Assignments to ghost variables can be combined atomically with other instructions.

Invariants and bounded delay

For us a predicate is a boolean function on the global state. A predicate is called invariant iff it holds in all reachable states. It is called stable (or inductive) iff it is preserved by every transition. Clearly, a stable predicate that holds initially is an invariant, and every predicate implied by an invariant is an invariant. Thus, the standard way to prove that a predicate P is invariant is to find a stable predicate Q that holds initially and that implies P.

In practice, invariants are obtained in the following way. Let us write $P \triangleright Q$ to denote that every transition that starts in a state where P holds terminates in a state where Q holds (all transitions terminate). Now, let $(i :: P_i)$ be a family of predicates that all hold initially. Let R be the conjunction of all P_i. Assume that $R \triangleright P_i$ holds for all i. Then predicate R is stable, holds initially, and implies all P_i. Therefore each P_i is an invariant. In such a construction, the predicates P_i are called the *constituent* invariants, while other predicates implied by R are called *derived* invariants.

In (Bjøerner *et al.* 1997, Manna *et al.* 1994) a more or less dual approach is advocated: there the global invariant is regarded as a disjunction of predicates. We do not know whether that approach could be used in our algorithm.

If a concurrent system has more than one process, there are execution sequences in which not all processes act. Usually one disallows such execution sequences by requiring some form of justness or fairness, cf. (Manna *et al.* 1994). For this purpose, we use a finitary form of fairness which is called *bounded delay* and in which progress is specified for families of processes.

It is formalized as follows. A history of the system is a finite nonempty sequence of pairs (x_i, q_i), such that each x_i is a global state and q_i is a process name, and that, for each state x_i except the last one, the current instruction of q_i can transform x_i into x_{i+1}. The corresponding sequence of processes is called the schedule of the history.

A schedule is called a *round* for client process p iff it contains p and at least one member of each family of server processes (at least once). The schedule is said to be k–fair for client p iff it is a concatenation of k rounds for p. Client p is said to be served with delay $\leq\ k$ iff every history k–fair for p contains at least one completion of a transaction of p. We define *bounded delay for all client processes* as the existence of a number k, such that every client is served with delay $\leq\ k$.

By this definition, bounded delay implies progress for every active client as long as each family of server processes contains at least one active member. Actually, in this paper we do not prove bounded delay but we present a design in which further choices can lead to bounded delay, cf. (Hesselink 1998).

Computer aided design

We use the general purpose theorem prover Nqthm to verify whether predicates are invariant under the step relation. Since our processes can make many different steps, even the effect of steps on variables requires verification. Since we have many predicates and many variables, automation of the verification is essential. The automation does not directly generate invariants, but whenever the prover is not able to prove invariance of some predicate, its failing final proof obligation often suggests a useful auxiliary invariant. The method to use Nqthm is described in more detail in (Hesselink 1996). The present paper seems to indicate that the method scales. The `event` files for our proofs have a length of about 9000 lines. They can be inspected at our Web site. During the design, we often had to modify the list of invariants or the list of instructions. To ease consistent modification in large text files, we have given the constituent invariants names of the form (Xqn) where n is a number smaller than the pc values used.

The computer aided verification in (Sipma *et al.* 1996) seems to require less user interaction. On the other hand, the example shown in (Sipma *et al.* 1996) is very small in comparison with the system described below.

3 CONCURRENT DATA OBJECTS IN SHARED MEMORY

A data object has an internal state x which is modified by means of invocations u. So, formally, it is a tuple $\langle X, x_0, U, R\rangle$ where X is the state space of the data object, $x_0 \in X$ is the initial state, U is the input space (the set of invocations), and $R \subseteq X \times U \times X$ is the transition relation. If the object is invoked in state x with invocation u, it may go into state y iff $\langle x, u, y\rangle \in R$.

In (Hesselink 1995), we used a model in which the data object gives output at every invocation. Here, we just assume that the whole new state of the object is the output. This simplification makes no difference for the algorithmic aspects.

Just as in (Hesselink 1995), we assume that relation R is total, i.e., for every pair $\langle x, u\rangle$ there exists y with $\langle x, u, y\rangle \in R$. The new state y need not be unique, however, as was needed before.

We assume that there are a number of concurrent processes that have private variables and that communicate by means of shared variables. The data object resides in this shared data space. It is passive, and all its actions are performed by processes.

The concurrent data object $\langle X, x_0, U, R\rangle$ consists of a procedure that, conceptually, acts on one shared program variable `x` of type X and that could be specified by

$$\begin{array}{l} \mathbf{proc}\ \mathit{apply}\ (\mathbf{in}\ p : \mathit{process}, u : U;\ \mathbf{out}\ y : X) \\ \quad \{\ \mathbf{pre}\ \mathtt{x} = w,\ \mathbf{post}\ \mathtt{x} = y\ \wedge\ \langle w, u, y\rangle \in R\ \} \end{array}$$

for every value $w \in X$. A client process p calls procedure *apply* in the form $\mathit{apply}(p, u, y)$ for the treatment of invocation u to obtain the new state y. So, p and u are input parameters and y is a result parameter. All clients may call *apply* concurrently and repeatedly.

The aim is to construct a distributed implementation of *apply* by means of a given sequential implementation of procedure

$$\begin{array}{l} \mathbf{proc}\ \mathit{locapply}\ (\mathbf{in}\ u : U, w : X;\ \mathbf{out}\ y : X) \\ \quad \{\ \ \mathbf{post}\ \langle w, u, y\rangle \in R\ \} \end{array}$$

This procedure can be used by the processes, provided that concurrent reading and writing of variables of types U and X is avoided. More precisely, we assume that such variables reside in regular registers, see Section 9.

We formalize linearization in the following way, cf. (Hesselink 1995). For each client p, we define $\beta.p$ to be the list of consecutive pairs $\langle u, y\rangle$ of invocations and results of p. For a list β and element w, let $w : \beta$ be the list obtained by prefixing list β with w. Conceptually, each client executes the infinite loop

$$*\ [\ u := \mathit{arbitrary}\ ;\quad \mathit{apply}\ (\mathit{self}, u, y)\ ;\quad \beta :=\ \langle u, y\rangle : \beta\ ]\ .$$

We represent the logical history of the *object* by an ordered list σ of triples $\langle p, u, y\rangle \in P \times U \times X$, where P is the set of clients. The occurrence of $\langle p, u, y\rangle$ means that process p has performed an invocation u with resulting state y. Let ε be the empty list. In order to relate σ to β, we define $\sigma|p$ to be list of pairs given by $\varepsilon|p = \varepsilon$, and

$$(\langle q, u, y\rangle : \sigma)|p \quad = \quad \mathbf{if}\ p = q\ \mathbf{then}\ \langle u, y\rangle : (\sigma|p)\ \mathbf{else}\ \sigma|p\ \mathbf{fi}\ .$$

List σ is said to be acceptable iff it corresponds to a legal sequential history of the object. This is formalized in predicate *acc*, defined by $\mathit{acc}.\varepsilon = \mathit{true}$ and

$$\mathit{acc}.(\langle p, u, y\rangle : \sigma) \quad = \quad \mathit{acc}.\sigma \quad \wedge \quad \langle \mathit{stafi}.\sigma, u, y\rangle \in R$$

where $\mathit{stafi}.\sigma$ is the last state of history σ, defined by $\mathit{stafi}.\varepsilon = x_0$ and $\mathit{stafi}.(\langle p, u, y\rangle : \sigma) = y$.

The data object is said to be *linearizing* iff one can construct a ghost variable σ with initially $\sigma = \varepsilon$, that for every execution satisfies the invariants

(Lin0) $\quad acc.\sigma$;
(Lin1) $\quad \beta.q = \sigma|q$ for every $q \in P$, whenever client q is not in *apply*.

4 A LINEARIZING DESIGN

We model the repeated calls of procedure *apply* by means of a number of looping sequential processes. We use an instruction pointer *pc* for each process and we number the atomic commands. In this section we propose three families of processes, as yet without proof. Compare (Herlihy 1991, Hesselink 1995).

We use two regions of shared memory, one for the invocation values $u : U$, and one for the state values $x : X$. The pointers into these regions are called *addresses* and *locations*, respectively. In both cases we use the value 0 as the *nil* address; nothing is stored there.

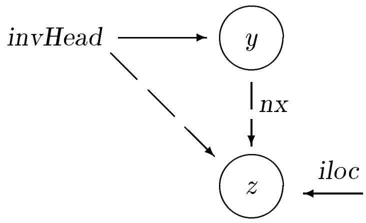
The following shared variables occur: *iloc.p* is the shared address of the current invocation of process p, and *inv.i* is the invocation stored at address i. The boolean *tolin.i* is a flag to indicate that the invocation at i is ready to be included in the linearization. Variable *post.i* gets the location of the new state corresponding to the invocation stored at i, and *sta.u* is the state located at u. We use *own.i* as a ghost variable to indicate the owner of an invocation address. The variables *iloc.i* and *own.i* will get nonzero values in collector processes, to be treated later. For the moment we only reset these variables.

Every client process executes the program Client. It has the private variables u for its current invocation, i and *sl* as copies of shared information, and the ghost variable β mentioned above. The elementary commands are numbered from 20 to enable easier modification. Notice that an invoking client q waits at command 21 until *iloc.q* is nonzero, and that it waits at 24 until *post.i* is nonzero. The private variable i is used to avoid that a single instruction has to access more than one shared variable. The result of the invocation is obtained in the read action 25.

Client
20 $\quad u := \mathit{arbitrary}$;
21 $\quad i := \mathit{iloc.self}$; **if** $i = 0$ **then goto** 21 **fi** ;
22 $\quad \mathit{inv}.i := u$;
23 $\quad \mathit{tolin}.i := \mathit{true}$;
24 $\quad \mathit{sl} := \mathit{post}.i$; **if** $\mathit{sl} = 0$ **then goto** 24 **fi** ;
25 $\quad \beta := \langle u, \mathit{sta.sl}\rangle : \beta$;
26 $\quad \mathit{iloc.self} := 0$; $\mathit{own}.i := 0$; **goto** 20 .

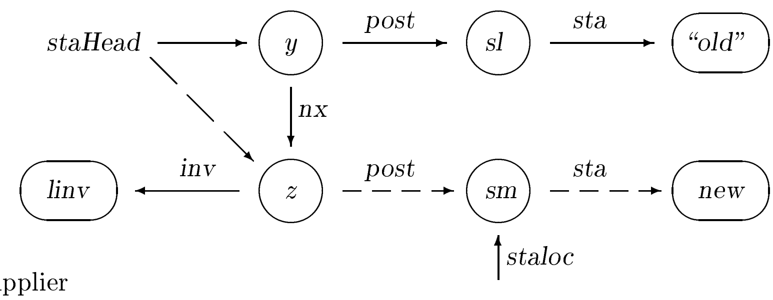
We introduce a family Linearizer of server processes for the linearization of the invocations. Ignoring all questions of fairness, each linearizer process chooses an arbitrary client and tries to linearize its invocation. We form a linked list with the shared variable nx as a forward pointer. Linearization of an address z means the inclusion of z at the head of this list. The shared variable $invHead$ points to the address most recently linearized. The linearizer processes use a private variables s for processes, and y and z for addresses.

```
Linearizer
30    choose s ∈ Client ;
31    y := invHead ;
32    z := iloc.s ;
33    if ¬tolin.z then goto 38 fi ;
34    if nx.y = 0 then nx.y := z fi ;
35    z := nx.y ;
36    tolin.z := false ;
37    if invHead = y then invHead := z fi ;
38    goto 30 .
```



The atomic commands 34 and 37 show that the shared variable $nx.y$ is a consensus register and that $invHead$ is a compare and swap register.

We introduce a family Applier of server processes that concurrently compute and store the results of procedure *locapply* for relevant invocations. We use an array *staloc* of locations for new states. The situation of Applier is sketched in the following diagram.




```
Applier
45    sm := staloc.self ;   if sm = 0 then goto 45 fi ;
46    y := staHead ;
47    z := nx.y ;   if z = 0 then goto 46 fi ;
48    linv := inv.z ;
49    sl := post.y ;
50    locapply(linv, sta.sl, new) ;
51    sta.sm := new ;
52    if post.z = 0 then
          post.z := sm ;   σ := ⟨own.z, linv, new⟩ : σ   fi ;
53    if staHead = y then staHead := z fi ;
```

```
54        if  post.z ≠ sm  then goto  56  fi ;
55        staloc.self := 0 ;
56        goto 45 .
```

An applier first waits until a free location *sm* is obtained. Then y becomes the address of the invocation most recently treated, and z becomes the address of the next invocation. The applier waits if there is no next invocation. Then *linv* gets the value of the new invocation and *sl* becomes the location of the (presumed) most recent state. Procedure *locapply* yields a candidate for the new state, which is then stored at *sm*. Lines 52, 53 of Applier can be compared with line 37 of Client. The assignment to $post.z$ is accompanied by an update of the ghost variable σ. This update must occur in command 52, since command 52 makes the new result available to the invoking process, see command 24 of Client. In 55, the location *staloc.self* is released, if necessary.

5 LINEARIZATION UNDER APPLIER

Up to this point nothing formal has been done. We have presented code for three families of processes, accompanied by informal arguments, as inspired by our previous experience in waitfree linearization. We now come to the central question: how to prove formally that the code satisfies the specification, i.e., the two invariants (Lin0) and (Lin1)?

We proceed as follows. We start with the invariants (Lin0) and (Lin1) of the specification, and analyse how these invariants can be invalidated. We then postulate new invariants as weak as possible to preserve the invariants. We analyse the new invariants in the same way, etc. The theorem prover is the central tool of the analysis: we submit invariants to the prover and see at which points additional arguments are needed.

We thus use the proof obligation (Lin0) as our first invariant:

(Aq0) $\quad acc.\sigma$.

Let us first prove that predicate (Aq0) is preserved by all actions of applier processes. In the course of this proof we postulate other invariants that must also be preserved by all applier actions. For the moment we ignore actions of client and linearizer processes.

First some terminology. A predicate Q is said not to be threatened by action m iff, for every global state that satisfies Q, the action of any process p with $pc.p = m$ preserves Q. If Q is threatened by action m, we need an additional invariant in the precondition to guarantee that Q is not falsified at m. Usually, the theorem prover is able to prove preservation of an invariant at all points where it is not threatened. Of course, the prover always fails where the invariant is threatened. It is our experience that it saves user time to anticipate most of the threats and prepare the remedies beforehand. For a private variable v, we write $v.p$ to denote the value of variable v of process p. We write $pc.p$ to denote the instruction pointer of process p.

The invariant (Aq0) is threatened only by the update of σ in command 52. In order to prove that (Aq0) is preserved at 52, it suffices to postulate the invariants

$$
\begin{array}{ll}
\text{(Aq1)} & 50 < pc.q \leq 52 \quad \Rightarrow \quad \langle sta.(sl.q), linv.q, new.q \rangle \in R \ ; \\
\text{(Drv0)} & pc.q = 52 \quad \wedge \quad post.(z.q) = 0 \quad \Rightarrow \quad sta.(sl.q) = stafi.\sigma \ .
\end{array}
$$

Remark. Many of our invariants are of the form $b < pc.q \leq e \Rightarrow X$. We use intervals open to the left, since the command at b usually verifies X or makes it true, while the command at e is allowed to make X false. □

We decide to let predicate (Drv0) be a derived invariant. In order to get (Drv0), we postulate the new constituent invariants

$$
\begin{array}{ll}
\text{(Aq2)} & 47 < pc.q \leq 56 \quad \Rightarrow \quad z.q = nx.(y.q) \ ; \\
\text{(Aq3)} & 49 < pc.q \leq 56 \quad \Rightarrow \quad sl.q = post.(y.q) \ ; \\
\text{(Aq4)} & 46 < pc.q \leq 52 \quad \wedge \quad post.(nx.(y.q)) = 0 \quad \Rightarrow \quad y.q = staHead \ ; \\
\text{(Aq5)} & stafi.\sigma \ = \ \mathbf{if} \ \ post.(nx.staHead) = 0 \ \ \mathbf{then} \ \ sta.(post.staHead) \\
 & \qquad\qquad\quad \mathbf{else} \ \ sta.(post.(nx.staHead)) \ \ \mathbf{fi} \ .
\end{array}
$$

Indeed, it is easy to verify that these four predicates together imply (Drv0).

Now we have to prove the invariance of (Aq1) through (Aq5) under applier processes. The invariance of (Aq1) is rather easy. First note that (Aq1) is preserved when process q itself executes command 50 because of the specification of procedure *locapply*. It follows that, since *linv*, *sl*, and *new* are private variables, (Aq1) is threatened only when some process p executes 51 with $sm.p = sl.q$. This is a question of the avoidance of interference. Indeed, preservation of (Aq1) at 51 follows from (Aq3) together with the new postulate

$$
\text{(Ia0)} \qquad 45 < pc.q \leq 52 \quad \Rightarrow \quad sm.q \neq post.k \ .
$$

Here k stands for an arbitrary address, possibly 0. Indeed all invariants are universally quantified over processes q (and r) and addresses k (and m).

We treat (Ia0) as a derived invariant, which is implied by the new postulates

$$
\begin{array}{ll}
\text{(Aq6)} & 45 < pc.q \leq 57 \quad \Rightarrow \quad sm.q \neq 0 \ ; \\
\text{(Aq7)} & 45 < pc.q \leq 55 \quad \Rightarrow \quad sm.q = staloc.q \ ; \\
\text{(Aq8)} & staloc.q = post.k \neq 0 \quad \Rightarrow \quad pc.q \in \{53, 54\} \ .
\end{array}
$$

Here we allow pc values that do not yet occur. These pc values will occur later, and we want to avoid unnecessary modifications of the invariants as much as possible. Of course we have to replay the mechanical proof for every modification of the algorithm.

Predicate (Aq2) is not threatened by applier processes. Predicate (Aq3) is threatened only at 52. It is preserved at 52 if we postulate

(Aq9) $\quad 46 < pc.q \leq 56 \quad \Rightarrow \quad post.(y.q) \neq 0$.

Predicate (Aq4) is preserved at 53 because of (Aq2) and the new postulate

(Aq10) $\quad 52 < pc.q \leq 56 \quad \Rightarrow \quad post.(z.q) \neq 0$.

Predicate (Aq5) is preserved at 51 because of (Ia0). It is preserved at 52 and 53 because of (Aq2), (Aq4), (Aq6), (Aq10), and the new postulates

(Aq11) $\quad pc.q = 52 \quad \Rightarrow \quad new.q = sta.(sm.q)$;
(Aq12) $\quad post.(nx.(nx.staHead)) = 0$.

The predicates (Aq6) and (Aq7) are not threatened. The predicates (Aq8) and (Aq9) are preserved because of (Aq7) and the new postulates

(Bq0) $\quad staloc.q = staloc.r \neq 0 \quad \Rightarrow \quad q = r$;
(Bq1) $\quad staloc.q = post.k \neq 0 \quad \Rightarrow \quad z.q = k$;
(Bq2) $\quad post.staHead \neq 0$.

Predicate (Aq10) is preserved because of (Aq6), used at 52. Preservation of (Aq11) only requires (at 51) the new predicate

(Ia2) $\quad 45 < pc.q \leq 54 \quad \wedge \quad 45 < pc.r \leq 54 \quad \wedge \quad sm.q = sm.r \quad \Rightarrow \quad q = r$.

This predicate follows from (Aq6), (Aq7), and (Bq0). Predicate (Aq12) is preserved because of (Aq2), (Aq4), and the new postulates

(Bq3) $\quad 47 < pc.q \leq 56 \quad \Rightarrow \quad z.q \neq 0$;
(Bq4) $\quad nx.k = k \quad \Rightarrow \quad k = 0$;
(Bq5) $\quad post.k = 0 \quad \Rightarrow \quad post.(nx.k) = 0$.

For preservation of (Bq5) we also need the postulate

(Bq6) $\quad nx.k = nx.m \neq 0 \quad \Rightarrow \quad k = m$.

Using all current invariants, one can prove preservation of (Bq0) up to (Bq6).

Remarks. The choice of invariants constrains later stages of the design. Some invariants are more or less forced, like (Aq1). The choice of (Aq2) is not forced, however, and will preclude the garbage collector to reset $nx.k := 0$ while $k = y.q$. It is conceivable that this invariant can be avoided. The invariants (Bq5) and (Bq6) also impose considerable constraints on garbage collection.

As an alternative for (Aq12) and (Bq5), one may introduce nx^k for repeated application of nx and then postulate that $post.(nx^k.staHead) = 0$ for every $k \geq 2$. We avoid this since mechanical verification of such an invariant requires much user assistence. □

6 PRESERVATION UNDER THE ACTIONS OF LINEARIZER

We now have to ensure that the invariants postulated in the previous section are also preserved by the action of linearizer and client processes. We first treat linearizers.

It is easy to see that the invariants of the families (Aq) and (Bq) can only be threatened by the commands 32, 34 and 35. Indeed, predicate (Aq2) is threatened at 34, and preserved because of (Bq3). The predicates (Aq4) and (Aq5) are threatened by 34 and preserved if we postulate

(Cq0) $\quad post.0 = 0 \;;$
(Cq1) $\quad pc.q = 34 \quad \wedge \quad nx.(y.q) = 0 \quad \Rightarrow \quad post.(z.q) = 0 \;.$

The invariants (Aq6), (Aq7), (Aq8), (Aq9), (Aq10), (Aq11), (Bq0), (Bq2), (Bq3) need no new arguments. Predicate (Aq12) is preserved because of (Bq5) and (Cq1). Predicate (Bq1) is preserved at 32 and 35 because of (Aq8). Predicate (Bq4) is threatened at 34, and preserved because of the new postulates

(Cq2) $\quad 31 < pc.q \leq 37 \quad \wedge \quad nx.(y.q) = 0 \quad \Rightarrow \quad y.q = invHead \;;$
(Cq3) $\quad pc.q = 34 \quad \wedge \quad nx.(y.q) = 0 \quad \Rightarrow \quad tolin.(z.q) \;;$
(Cq4) $\quad \neg tolin.invHead \;.$

Predicate (Bq5) is preserved at 34 because of (Cq1). Predicate (Bq6) is threatened at 34, and preserved because of (Cq2), (Cq3), and the new postulate

(Cq5) $\quad tolin.(nx.k) \quad \Rightarrow \quad invHead = k \;.$

Now the new postulates must also be proved to be invariant, also under actions of applier processes. Preservation of (Cq0) only requires (Bq3), which is used at 52. Preservation of (Cq1) at 52 follows from (Aq2), (Bq3), (Cq2), (Cq3), and (Cq5). Preservation of (Cq1) at 33 follows (with $k = z.q$) from the new postulate

(Cq6) $\quad tolin.k \quad \Rightarrow \quad post.k = 0 \quad \vee \quad nx.invHead = k \;.$

Preservation of (Cq2) follows from the new postulates

(Cq7) $\quad 34 < pc.q \leq 37 \quad \Rightarrow \quad nx.(y.q) \neq 0$;
(Cq8) $\quad 35 < pc.q \leq 37 \quad \Rightarrow \quad nx.(y.q) = z.q$.

For preservation of (Cq6) at 37 and (Cq7) at 34, we postulate

(Cq9) $\quad pc.q = 37 \quad \Rightarrow \quad \neg tolin.(z.q)$;
(Cq10) $\quad \neg tolin.0$.

For preservation of (Cq8) at 34 we need (Cq7). Preservation of (Cq9) and (Cq10) under Linearizer and Applier is trivial.

Remark. Invariant (Cq6) came as a surprize. We had expected $tolin.k \Rightarrow post.k = 0$, but that predicate is too strong and can be falsified at command 52. □

7 PRESERVATION UNDER CLIENT

We turn to the actions of client processes. The only threat of Client to the above invariants is the modification of $tolin.i$ in 23, and this is only a threat for the invariants (Cq4), (Cq5), (Cq6), (Cq9), and (Cq10). In order to preserve (Cq4) and (Cq5) under action 23, we postulate

(Dq0) $\quad 21 < pc.q \leq 26 \quad \Rightarrow \quad iloc.q = i.q$;
(Dq1) $\quad 18 < pc.q \leq 23 \quad \Rightarrow \quad iloc.q \neq invHead$;
(Ia1) $\quad 21 < pc.q \leq 23 \quad \Rightarrow \quad i.q \neq nx.k$.

Predicate (Ia1) follows from (Dq0) and the new postulates

(Dq2) $\quad 18 < pc.q \leq 23 \ \wedge \ iloc.q = nx.k \quad \Rightarrow \quad iloc.q = 0$;
(Dq3) $\quad 21 < pc.q \leq 26 \quad \Rightarrow \quad i.q \neq 0$.

Predicate (Cq6) is preserved at 23 because of (Dq0) and the new postulate

(Dq4) $\quad 18 < pc.q \leq 23 \quad \Rightarrow \quad post.(iloc.q) = 0$.

Predicate (Cq9) is preserved at 23 because of (Cq8) and (Ia1). Preservation of (Cq10) follows from (Dq3).

In order to preserve (Dq1) and (Dq2), we postulate

(Dq5) $\quad invHead \neq 0$;
(Dq6) $\quad 18 < pc.q \leq 23 \quad \Rightarrow \quad \neg tolin.(iloc.q)$.

Now one can prove the invariance of (Dq0) up to (Dq5). Predicate (Dq6) also needs the new postulate

(Ia3) $\quad iloc.q = iloc.r \neq 0 \quad \Rightarrow \quad q = r$,

which will follow from a postulate in the next section. This concludes the proof of invariance of (Aq0), i.e., proof obligation (Lin0).

8 THE TREATMENT OF THE CLIENT PROCESSES

We now turn to proof obligation (Lin1), i.e., $\beta.q = \sigma|q$ for every $q \in P$ whenever client q is not in *apply*. This expresses that the client processes are served correctly. We formalize and strengthen proof obligation (Lin1) to

(Eq0) $$\beta.q \;=\; \textbf{if}\;\; post.(iloc.q) = 0 \;\;\vee\;\; pc.q = 26 \;\; \textbf{then}\;\; \sigma|q \;\; \textbf{else}\;\; tail.(\sigma|q) \;\; \textbf{fi} \; .$$

Predicate (Eq0) is clearly threatened when some process executes command 52. In this command we use the ghost variable *own.z* to indicate the owner of the invocation at address z. To formalize ownership, we postulate

(Eq1) $\quad own.k = 0 \quad \vee \quad iloc.(own.k) = k$;
(Eq2) $\quad iloc.q = 0 \quad \vee \quad own.(iloc.q) = q$.

Notice that (Ia3) follows from (Eq2).

We now come at a tricky point. By convention, we disallow process number 0. So a client with number 0 does not exist. Yet we want (Eq0) to hold for $q = 0$, since $\beta.0 = \sigma|0$ together with $\beta.0 = \varepsilon$ expresses that no ownerless invocations have been treated. Moreover, $own.k = 0$ must be allowed, just like $iloc.q = 0$. Therefore, in (Lin1), we take $P = \{0\} \cup Client$ and postulate the typing invariant $own.k \in P$.

For preservation of (Eq0) at 52, we postulate

(Drv1) $$pc.q = 52 \;\;\wedge\;\; post.(z.q) = 0 \;\;\wedge\;\; r = own.(z.q) \;\;\Rightarrow\;\; r \neq 0 \;\;\wedge\;\; z.q = iloc.r \;\;\wedge\;\; pc.r \neq 26 \; .$$

Preservation of (Eq0) at 52 follows from (Aq6), (Bq3), (Eq2), and (Drv1). Predicate (Eq0) is further threatened only when process q itself executes commands 25 or 26. Preservation at 26 follows from (Cq0). Preservation at 25 follows from the new postulates

(Eq3) $\quad 24 < pc.q \leq 26 \quad \Rightarrow \quad post.(iloc.q) \neq 0$;
(Drv2) $\quad pc.q = 25 \quad \Rightarrow \quad head.(\sigma|q) = \langle q, u.q, sta.(sl.q)\rangle$.

The predicates (Drv1) and (Drv2) follow from (Aq2), (Bq3), (Dq2), (Eq1), (Eq3), and the new postulates

(Eq4) $\quad post.(nx.k) = 0 \quad \wedge \quad own.(nx.k) = 0 \quad \Rightarrow \quad nx.k = 0$;
(Eq5) $\quad post.(iloc.q) \neq 0 \ \Rightarrow \ head.(\sigma|q) = \langle q, u.q, sta.(post.(iloc.q))\rangle$;
(Eq6) $\quad pc.q = 25 \quad \Rightarrow \quad sl.q = post.(iloc.q)$.

We turn to the preservation of (Eq1) through (Eq6). The predicates (Eq1), (Eq2), (Eq3) need no new arguments. To preserve (Eq4) at 34 and (Eq5) at 52, we postulate

(Eq7) $\quad post.k = 0 \quad \wedge \quad own.k = 0 \quad \Rightarrow \quad \neg tolin.k$;
(Eq8) $\quad 48 < pc.q \leq 52 \quad \Rightarrow \quad linv.q = inv.(z.q)$;
(Eq9) $\quad q \in Client \quad \Rightarrow \quad 18 < pc.q \leq 22 \quad \vee \quad u.q = inv.(iloc.q)$.

We need no new postulates to prove preservation of (Eq6) up to (Eq9). This concludes the treatment of proof obligation (Lin1).

Remark. We could have chosen to strengthen (Eq4) and (Eq7) to $post.k = 0 \Rightarrow own.k \neq 0$, but then $post.k := 0$ must be done by Client, so that garbage collection may be hampered by “old” pointers $post.k \neq 0$. □

9 REGULAR REGISTERS

As announced above the invocation and state values of the object need not be so small that they can be put into an atomic register, i.e., one that allows concurrent reading and writing. A register is called *regular* if concurrent reading is allowed, but whenever a process is writing the register no other process is allowed to read or write the register.

We assume that the elements of the arrays *inv* and *sta* are regular registers. So, we have the proof obligations to show that, whenever some process, say q, is writing $inv.k$ or $sta.k$, no other process, say r, is writing or reading at that address or location. This amounts to the regularity requirements

(rg0) $\quad q \neq r \quad \wedge \quad pc.q = 22 \quad \wedge \quad pc.r = 22 \quad \Rightarrow \quad i.q \neq i.r$;
(rg1) $\quad pc.q = 22 \quad \wedge \quad pc.r = 48 \quad \Rightarrow \quad i.q \neq z.r$;
(rg2) $\quad q \neq r \quad \wedge \quad pc.q = 51 \quad \wedge \quad pc.r = 51 \quad \Rightarrow \quad sm.q \neq sm.r$;
(rg3) $\quad pc.q = 51 \quad \wedge \quad pc.r = 25 \quad \Rightarrow \quad sm.q \neq sl.r$;
(rg4) $\quad pc.q = 51 \quad \wedge \quad pc.r = 50 \quad \Rightarrow \quad sm.q \neq sl.r$.

These regularity requirements are all implied by the above invariants, in particular by (Ia0) up to (Ia3), and (Aq2), (Aq3), (Dq0), (Dq3), (Eq6).

10 GARBAGE COLLECTION

In view of space limitations we now lower the levels of formality and completeness considerably. So, for the remainder of the design we omit most of the invariants and proofs. The details can be found on our Web site.

For the sake of efficiency, we provide every address with sufficient information for a collector process to decide that the address is free for recycling, without sweeps over lists of processes or addresses. This information will consist of a number (*cnt*) and five essentially boolean flags.

The first requirement of progress is that an interested client p gets an address $i = iloc.p \neq 0$. For this purpose we use a command of the form

$$\textbf{if}\ \ iloc.sg = 0\ \ \textbf{then}\ \ iloc.sg := yg\ ;\quad own.yg := sg\ \ \textbf{fi}.$$

So we need to find addresses *yg* that can be used in this command. The theorem prover has been used to find the conditions on *yg* needed.

In order to avoid that the address y of a linearizer or applier process is recycled prematurely, an address k gets a counter for the number of such processes q with $y.q = k$. So we redefine

```
Linearizer
28        choose  s ∈ Client ;
29        y := invHead ;
30        cnt.y := cnt.y + 1 ;
31        if  y ≠ invHead  then goto 38 fi ;
32 ... 37 Instructions as before ;
38        cnt.y := cnt.y − 1 ;   goto 28 .
```

The test of $y \neq invHead$ in command 31 serves to handle the case that a collector grabs the address while a delayed linearizer is about to increment the counter from 0 to 1. A similar modification is made in Applier below.

In order to avoid that the address of a linearized invocation that has not yet been treated, is recycled prematurely, the addresses get boolean flags *usob* ("in use for the object") with the invariants that *usob* holds for the addresses *invHead* and *nx.invHead*, and for all addresses k where $post.k = 0$ holds. The flags are lowered at command 56. We thus redefine

```
Applier
43        sm := staloc.self ;   if  sm = 0  then goto 43  fi ;
44        y := staHead ;
45        cnt.y := cnt.y + 1 ;
46        if  y ≠ staHead  then goto 57 fi ;
47        z := nx.y ;   if  z = 0  then goto 56 fi ;
48 ... 55 Instructions as before ;
56        usob.y := false ;
57        cnt.y := cnt.y − 1 ;   goto 43 .
```

In the invariants (Ia0), (Aq6), and (Aq7), the starting point 45 of the ranges must now be replaced by 43. We can then replay the old proofs on the theorem prover. Note that extension at the end of the ranges was anticipated.

It is easy to see that the counters indeed count, i.e., that $cnt.k$ always equals the number of processes q with $y.q = k$, and $pc.q$ either in $(30\ ..\ 38]$ or in $(45\ ..\ 57]$. This is one of the harder invariants to prove with a theorem prover.

We also have to avoid recycling the address of the result of a delayed client. The test $own.k = 0$ would serve for this purpose. Since we want to treat *own* as a ghost variable, however, we give each address a flag *isil* (this stands for *is iloc*) with the intention that $own.k \neq 0$ implies $isil.k$. The idea is to set $isil.i$ to *false* after command 26, but we do it at 19, as was anticipated in the invariants (Dq2), (Dq4), etc. Thus, program Client is modified as follows.

Client
19 $\quad isil.i := false$;
20 ... 25 $\quad$ Instructions as before ;
26 $\quad iloc.self := 0$; $\quad own.i := 0$; $\quad$ **goto** 19 .

Collector processes

The collector processes have to inspect addresses and make these available to the clients. For each separate address, they must work under mutual exclusion. For this purpose we use a strong consensus object *down*.

We introduce a flag *open* to mark addresses ready for recycling and not yet used. This flag is raised at 78 and lowered at 75 just before recycling is attempted. We thus arrive at the following form of garbage collection

Collector
59 $\quad$ **choose** $yg \in Address$;
60 $\quad$ **if** $down.yg = 0$ **then** $down.yg := 1$ **else goto** 59 **fi** ;
61 $\quad$ **if** $open.yg$ **then goto** 75 **fi** ;
62 $\quad$ **if** $usob.yg$ **then goto** 79 **fi** ;
63 $\quad$ **if** $cnt.yg \neq 0$ **then goto** 79 **fi** ;
64 $\quad zg := nx.yg$; { decouple zg }
65 $\quad nx.yg := 0$;
66 $\quad isnx.zg := false$;
67 $\quad$ **if** $isil.yg$ **then goto** 79 **fi** ;
68 $\quad$ **if** $isnx.yg$ **then goto** 79 **fi** ;
69 $\quad usob.yg := true$;
70 $\quad ug := post.yg$; { decouple ug }
71 $\quad post.yg := 0$;
72 $\quad occ.ug := false$;
73 $\quad isil.yg := true$;
74 $\quad isnx.yg := true$;
75 $\quad open.yg := false$;

76 **choose** $sg \in \mathit{Client}$;
77 **if** $\mathit{iloc}.sg = 0$ **then** { recycle yg }
$\mathit{iloc}.sg := yg$; $\mathit{own}.yg := sg$; **goto** 79 **fi** ;
78 $\mathit{open}.yg := \mathit{true}$;
79 $\mathit{down}.yg := 0$; **goto** 59 .

We use a flag *isnx* at each address to record that it does not occur as *nx* of another address. After the tests in 61, 62, and 63, we let the successor of *yg* be decoupled and set its *isnx* to false. This form of garbage collection is possible because of (Bq6). After two subsequent tests the location of *yg* is decoupled, and this is recorded by lowering flag *occ*.

The order of the commands is critical. In particular, command 62 must come before the others, and 63 must be the next. The tests 67 and 68 must come before commands 69 through 74. It is good for progress that the tests 67 and 68 come after the decoupling of $nx.yg$.

Distributing locations

We finally introduce processes to distribute locations, i.e., to give *staloc* adequate nonzero values. The procedure is similar as for invocation addresses, but simpler since some work has been done in Collector. We use a shared variable *lock* to avoid that the same location is distributed by different distributors at the same time. We use the flags *occ*, which were introduced above to indicate that a location is occupied. In this way, we arrive at

Distributor
90 **choose** $ug \in \mathit{Location}$;
91 **if** $\mathit{lock}.ug = 0$ **then** $\mathit{lock}.ug := 1$ **else goto** 90 **fi** ;
92 **if** $\mathit{occ}.ug$ **then goto** 96 **fi** ;
93 $\mathit{occ}.ug := \mathit{true}$;
94 **choose** $sg \in \mathit{Applier}$;
95 **if** $\mathit{staloc}.sg = 0$ **then** $\mathit{staloc}.sg := ug$ **else goto** 94 **fi** ;
96 $\mathit{lock}.ug := 0$; **goto** 90 .

11 THE INITIAL STATE

The system we have developed is a reactive system, which is supposed never to terminate. Yet, after building it, one must be able to start it. We therefore now describe the initial state. All invariants are implied by the following predicate.

$$\begin{array}{l} \sigma = \varepsilon \quad \wedge \quad \mathit{staHead} = \mathit{invHead} \neq 0 \quad \wedge \quad \mathit{post}.\mathit{staHead} \neq 0 \\ \wedge \quad \mathit{sta}.(\mathit{post}.\mathit{staHead}) = x_0 \quad \wedge \quad \mathit{occ}.(\mathit{post}.\mathit{staHead}) \\ \wedge \quad \neg \mathit{open}.\mathit{staHead} \quad \wedge \quad \neg \mathit{isnx}.\mathit{staHead} \quad \wedge \quad \neg \mathit{isil}.\mathit{staHead} \\ \wedge \quad \mathit{post}.0 = 0 \quad \wedge \quad \neg \mathit{open}.0 \quad \wedge \quad \beta.0 = \varepsilon \quad \wedge \quad \mathit{iloc}.0 = 0 \end{array}$$

$$\begin{array}{ll} \wedge & (\forall\, k :: nx.k = 0 \ \wedge\ own.k = 0 \ \wedge\ cnt.k = 0 \ \wedge\ usob.k \ \wedge\ \neg tolin.k) \\ \wedge & (\forall\, k : k \neq 0 \ \wedge\ k \neq staHead : post.k = 0 \ \wedge\ open.k \ \wedge\ isnx.k \ \wedge\ isil.k) \\ \wedge & (\forall\, q :: \beta.q = \varepsilon \ \wedge\ iloc.q = 0 \ \wedge\ staloc.q = 0 \\ & \quad \wedge\ \ pc.q \in \{20, 28, 43, 59, 90\} \ \wedge\ (q \in Client \Rightarrow pc.q = 20)\,). \end{array}$$

12 CONCLUDING REMARKS

We have constructed a concurrent system for linearization of an arbitrary data object under a form of fault tolerance. The memory requirements are linear in the number of processes. We would not have been able to conclude the design without a powerful theorem prover to verify the invariants. The design has 61 atomic instructions in tightly coupled concurrency. Although we had previous experience with similar algorithms, we started from scratch and essentially concluded it within ten weeks. This indicates that the method of (Hesselink 1996) scales to intricate distributed algorithms of moderate size.

REFERENCES

Bjøerner, N., Browne, A. and Manna, Z. (1997) Automatic generation of invariants and intermediate assertions. Theor.Comp.Science **173**, 49–87.

Boyer, R.S. and Moore J S. (1988) A Computational Logic Handbook. Academic Press, Boston etc.

Fischer, M.J., Lynch, N.A. and Paterson, M.S. (1985) Impossibility of distributed consensus with one faulty process. J. ACM **32**, 374–382.

Herlihy, M.P. (1991) Wait–free synchronization. ACM Trans. on Program. Languages and Systems **13**, 124–149.

Herlihy, M.P. and Wing, J. (1990) Linearizability: a correctness condition for concurrent objects. ACM Trans Program Lang Syst **12**, 463–492.

Hesselink, W.H. (1995) Wait–free linearization with a mechanical proof. Distr. Comput. **9**, 21–36.

Hesselink, W.H. (1996) NQTHM proving sequential programs. Computing Science Reports CS-R9605, Groningen. See also our Web site.

Hesselink, W.H. (1998) A formal proof of fault tolerant progress. In preparation.

Manna, Z. and Pnueli, A. (1994) Temporal verification diagrams. In *Proc. Int. Symp. on Theoretical Aspects of Computer Software.* LNCS 789. Springer, pp. 726–765.

Owicki, S. and Gries, D. (1976) An axiomatic proof technique for parallel programs. Acta Informatica **6**, 319–340.

Sipma, H.B., Uribe, T.E. and Manna, Z. (1996) Deductive model checking. In *Proc. 8th Intl. Conference on Computer Aided Verification.* LNCS 1102, Springer, pp. 208–219.

13 BIOGRAPHY

Wim. H. Hesselink received his PhD in mathematics from the University of Utrecht in 1975. After ten years of research in algebraic groups and Lie algebras, he turned to computing science. His research interests include predicate transformation semantics of recursive procedures with nondeterminacy of various flavours, distributed programming, design and correctness of algorithms, and the use of mechanical theorem provers. He is a professor at Groningen University since 1994. He now just concluded a term as Chairman of the Department of Computing Science.