Distrib Comput (1995) 9: 21–36

DISTRIBUTED
COMPUTING
© Springer-Verlag 1995

Wait-free linearization with a mechanical proof

Wim H. Hesselink*

Department of Mathematics and Computing Science, Rijksuniversiteit Groningen, Postbox 800, 9700 AV Groningen, The Netherlands

Received: October 1993 / Accepted: February 1995

Wim H. Hesselink received his Ph.D. in mathematics from the University of Utrecht in 1975. After ten years of research in Lie algebras he turned to computer science. In 1986/1987 he was on leave with the Department of Computer Sciences of the University of Texas at Austin. Currently, he is chairman of the Department of Computing Science at the University of Groningen. His research interests include distributed programming, design and correctness of algorithms, mechanical-theorem proving, and predicate transformation semantics of recursive procedures with nondeterminacy of various flavors.

Summary. The correctness of a program for wait-free linearization of an arbitrary shared data object in bounded memory is verified mechanically. The program uses atomic read-write registers, an array of consensus registers and one compare and swap register. In the program, a number of processes concurrently inspect and modify a pointer structure without waiting. Consequently, the proof of correctness is very delicate. The theorem prover NQTHM of Boyer and Moore has been used to mechanically certify the correctness.

Key words: Linearizable – Shared data object – Consensus – Wait-free – Memory management – Correctness – Invariant – Grain of atomicity – Mechanical verification – Boyer Moore logic

1 Introduction of the project

In [16], we presented an algorithm for wait-free linearization of an arbitrary shared data object in bounded memory. The algorithm was provided with a complicated proof of correctness that needed twenty-four invariants. In view of this complexity we felt the need of mechanical verification, even though we were fairly convinced of the validity of the proof.

We knew the logic of the theorem prover NQTHM of Boyer and Moore, cf. [7, 8], but we had no experience with the prover. In November 1992, we embarked on the project to try and verify the program of [16]. We started with a pilot project about the correctness of a trivial concurrent program without communication. This took three weeks. In March 1993, the proof of the program of [16] was essentially concluded. This proof was based on a hard-coded functional translation of the program. We spent several months more to modify the proof in such a way that the program together with the declaration of the program variables is given to the prover in a purely syntactic form. This transformation was done to get cleaner statements of results and more general applicability of the methods involved. The present paper is a report of this work.

The choice of NQTHM for the project was based on the following considerations. Both NQTHM of [8] and HOL of [11] had recently been made available in our department. So the choice was between these two. Since no higher order functions were involved in the program or its proof, the greater expressive power of HOL was not necessary. We knew that the logic of NQTHM was adequate for the task and, since we knew that logic, it was likely that the use of NQTHM would require a smaller initial investment. Finally, the accomplishments of NQTHM or rather of its group of users (e.g., cf. [17] and [6]) justify the choice of NQTHM, even if other provers might be better.

To say that we knew the logic does not mean that we knew how the prover could be most efficiently led to construct proofs. The Handbook [8] gives ample advice, but it requires experience to appreciate the advice. Initial modelling choices are very important, since they may affect the efficiency of the prover during the whole project. Actually, we used a kind of incremental design and we modified the basic set-up a number of times.

Of course, we had to take some initial hurdles. Perhaps, the main difficulty for the inexperienced user is that NQTHM must be guided by rather implicit means. For example, the order of the hypotheses of a lemma is taken by the prover as a hint how to choose instantiations in later applications of the lemma. During the project we got

* e-mail: wim@cs.rug.nl

invaluable assistance from J. Moore, one of the constructors of NQTHM. On a number of occasions, NQTHM seemed unable to make an obvious inference, but J. Moore was always willing and able to solve the problem on short notice.

We succeeded in verifying the algorithm. In general, the arguments used in [16] were sufficient, but it turned out to be convenient to rearrange the order of usage of the arguments. Additional arguments were needed only rarely. Occasionally, the use of the prover led to a better separation of concerns.

It turns out that the mechanical proof can be extended with not too much effort in such a way that the program has a finer grain of atomicity and needs a weaker precondition. This extension only requires a new traversal of a modification of the proof. Here, a mechanical proof has great advantages over a mental proof, especially when there are many case distinctions that can be affected by the modification. These advantages are also stressed in the conclusions of [24].

We regard the proof in [16] of progress of the algorithm as completely satisfactory. Yet, it turns out that the mechanical proof needs a new, and more formal, proof obligation. The ideas of the proof of [16] are still applicable. Unfortunately, we are now forced to provide explicit upper bounds. As a by-product, therefore, the project has led to a better understanding of the worst-case time complexity.

2 Correctness and mechanical verification

The problem we treat in this paper is a problem of the correct design of a distributed algorithm with shared memory. So it belongs to the field of programming methodology. This field seems to have got its momentum by the threatening possibilities of distributed algorithms, e.g., see the concluding remarks of [10]. Yet its results are mainly applied to sequential programs. The methods proposed for distributed programs in [3] and [23] are sufficiently strong but often lead to a horrible growth of proof obligations.

For this reason, methodologists and theoreticians have concentrated on developing alternative languages and specification formalisms, like CCS [20], CSP [18], Unity [9], ACP [5], action systems [4], to mention only a few. All these formalisms have merits, but the central problem of exploding complexity has not been solved.

In the present paper we use the old methods of [3] and [23], but we apply a mechanical theorem prover to control the complexity. We are not aware of applications of mechanical theorem provers to correctness proofs of distributed algorithms of this size and kind. Indeed, as suggested by one of the referees, the project may be regarded as a 'cutting edge' case study in the mechanical verification of concurrent algorithms. In the rest of this section we give a comparison with other mechanical verifications of algorithms.

The algorithm can be compared with the example treated by Russinoff in [25]. In that example, two processes concurrently act on three simple global variables with programs of four or five atomic actions. Our algorithm is definitely more complex: it allows an arbitrary number of processes, it uses eight shared arrays, each process has a number of private variables, the program of all processes has 32 atomic actions.

Of course, our algorithm is dwarfed by the work on the short stack, reported in [6], but there the complexity is due to the aim to verify a complete computer system, not a single algorithm.

Another referee suggested a comparison with [24]. That paper contains a clear introduction to and convincing arguments for machine verification of distributed and critical algorithms. The algorithm treated is the interactive convergence algorithm for fault-tolerant synchronization of clocks. It has roughly the same complexity as our algorithm, but the algorithm and the complexity issues involved are of a completely different nature. The algorithm is a mathematical procedure to choose adequate resynchronizations and the correctness conditions are arithmetical bounds on differences between estimates of time. The recent paper [22] also deals with asynchronous clocks. Its focus is entirely on the physical problems introduced by asynchrony, namely how clock rates, delay, and phase shift affect the received signal. Our algorithm, on the other hand, is a set of concurrently executed computer programs and the problems are due to the interleaved modifications of a shared pointer structure.

3 Introduction of wait-free linearization

We turn to the description of the problem.

A *concurrent data object* is a data structure shared by concurrent processes. The object must behave as if the invocations are processed in some sequential order. This requirement is formalized in the concept of linearizability (see [15], we come back to this in Sect. 5). Traditionally, linearizability is achieved by means of operations that temporarily block the progress of some processes. The disadvantage of such operations is that if a process is delayed (or stopped) other processes are delayed as well.

Therefore, recently, the concept of wait-free implementations has been proposed. A wait-free implementation of the concurrent data object is one in which every process completes its invocation in a bounded number of atomic actions, regardless of the actions and the execution speeds of the other processes, see [12, 14]. So, unbounded busy waiting loops or idle-waiting primitives are forbidden. There is no assumption that every process makes progress. A wait-free implementation is fault-tolerant in the sense that, if some process stops executing, the invocations of other processes are not affected.

One of the simplest concurrent data objects is the *atomic read-write register*: a shared variable, say $\mathtt{x}$, with the only atomic actions $\mathtt{x} := u$ and $z := \mathtt{x}$, for some private variables u and z. Notice that we use the convention that shared program variables are in typewriter font. Constants, private program variables, parameters and mathematical variables are in math-italic. From Sect. 8 onward, we shall also use typewriter font for all expressions in the language of the theorem prover.

It has been shown by Herlihy [14] that atomic read-write registers are not sufficient to construct wait-free implementations of many important data objects. In [1], on the other hand, it is shown that atomic read-write registers are sufficient to construct useful data objects like counters.

Another simple data object is the *consensus register*. This is a shared variable, say x, with an atomic read action $z := \mathbf{x}$ and an atomic setting action

$$\langle \mathbf{if}\ \mathrm{x} = 0\ \mathbf{then}\ \mathrm{x} := u\ \mathbf{fi} \rangle \tag{1}$$

where 0 is some constant of the same type as x, and z and u are private variables as before. If $\mathrm{x} \neq 0$, command (1) is equivalent to skip. There is also an atomic write operation $\mathrm{x} := 0$. Consensus registers are sometimes called logical variables or permanents, see e.g. [21] Sect. 1.4.

A slightly more powerful object is the *compare and swap register*. This is a shared variable x with an atomic read action $z := \mathrm{x}$ and an atomic setting action

$$\langle \mathbf{if}\ \mathrm{x} = u\ \mathbf{then}\ \mathrm{x} := v\ \mathbf{fi} \rangle \tag{2}$$

where z, u and v are private variables, cf. [14] p. 135.

Herlihy [12, 14] and Plotkin [26] have shown that wait–free implementations of arbitrary data objects can be constructed by means of atomic read–write registers together with consensus registers. Let n be the number of processes. The implementation of [26] requires memory of order n^2 and has a worst–case time complexity of order n^3. The implementation of [14] requires memory of order n^3, has a worst–case time complexity of order n^2 and, moreover, requires unbounded integers. The latter requirement is a serious drawback, since unbounded integers are not available in bounded space.

In [13], Herlihy presents a construction based on a compare and swap register with memory of order n^2 and worst–case time complexity of order n^2. For all three implementations, the outline provided is rather sketchy. This makes it hard to prove or to refute their correctness.

In [16], we presented a wait–free implementation of an arbitrary data object that requires memory of order n^2 and that has a worst–case time complexity of order n. The implementation uses a compare and swap register, just as in [13]. The paper [16] contains an assertional proof of correctness, but the proof is so complicated that it is not completely convincing. This observation was the starting point of the present project.

The project has resulted in an implementation closely related to the implementation of [16] but with a mechanical proof of correctness and a somewhat finer grain of atomicity. In particular, it satisfies the condition that each elementary command A may refer to at most one variable that can be changed by another process while A is being executed, cf. [23]. We have also been able to formalize and verify the safety of the non-atomic shared variables.

In a stricter sense, the result of the project is a file of definitions, lemmas and hints that, when loaded into NQTHM, yields proofs of all lemmas that are marked as proof obligation. This final verification can be performed by every observer with access to the file and to NQTHM. The file is named qspace.events and can be obtained by anonymous ftp from ftp.cs.rug.nl, in the directory /pub/boyer-moore.

4 Overview

In Sect. 5, we develop a formal description of the problem together with the list of proof obligations. In Sect. 6, we present a variation of our solution in [16]. This variation is a program with a weaker precondition and a finer grain of atomicity than the one of [16]. In Sect. 7, the program of Sect. 6 is reformulated to facilitate the proof. This reformulation consists of the addition of auxiliary history variables as required by the proof obligations, the addition of labels to discuss the execution of the processes, and the elimination of composite constructs for **if** and **while** by means of **goto** commands since these composite constructs are not executed atomically.

Section 8 contains a sketch of some relevant aspects of the theorem prover NQTHM, followed by a description how the program is modelled as a function that yields the new implementation state in terms of the old implementation state, the acting process and one other argument. Section 9 contains a description of the formal proof obligations, i.e., of the main theorems that are proved by the prover.

Section 10 gives a very short description of our experiences with the prover. Section 11 gives a description of the global structure of the proof. Section 12 describes the initial parts of the proof. Section 13 contains some examples of invariants and a sketch of how the invariance of these predicates is proved by the prover. Finally, in Sect. 14, we sketch the proof of wait-free progress. Some conclusions are drawn in Sect. 15.

5 Data objects and concurrency

A data object is a tuple $\langle X, U, Z, x_0, R \rangle$ where X is the state space of the object, $x_0 \in X$ is the initial state, U is the input space (the set of invocations), Z is the output space (the set of result values) and $R \subseteq X \times U \times X \times Z$ is the transition relation. If the object is invoked in state x with invocation u, it may go into state y and return the output z if and only if $\langle x, u, y, z \rangle \in R$.

Just as in [16], we assume that the object is *total* and *deterministic*, in the sense that in every state every invocation allows precisely one new state and precisely one result: for every pair $\langle x, u \rangle$ with $x \in X$ and $u \in U$, there is precisely one pair $\langle y, z \rangle$ with $\langle x, u, y, z \rangle \in R$. The requirement of totality (the existence of a resulting pair $\langle y, z \rangle$ for every pair $\langle x, u \rangle$) formalizes the assumption that no operation can be blocked. Determinacy is postulated for the sake of simplicity of the algorithm. This assumption is essential for the present algorithm, but a variation of the algorithm that avoids this assumption is in preparation.

We assume that there are n processes, with process identifiers of type *process*. A concurrent implementation of a data object $\langle X, U, Z, x_0, R \rangle$ is a procedure that, conceptually, acts on one global program variable x of type X and that could be specified by

proc *apply* (**in** P: *process*, u: U; **out** z: Z)

$\{\mathbf{pre}\ \mathrm{x} = w,\ \mathbf{post}\ \langle w, u, \mathrm{x}, z \rangle \in R\}$

Here, w is a logical variable that stands for the value of $\mathbf{x}$ in the precondition. Process P calls procedure *apply* in the form $apply(P, u, z)$ for the treatment of invocation u with result z. So P and u are input parameters and z is a result parameter.

All processes may call *apply* concurrently and repeatedly. During execution of such a call the process is said to be *invoking*. The data object itself is passive; the subcommands of *apply* are executed by the invoking process. Yet the implementation is required to be *linearizable*, in the sense that each call of *apply* appears to take effect instantaneously at some point between the invocation and the response. Linearizability implies that processes appear to be interleaved at the granularity of complete operations and that the order of non-overlapping operations is preserved. See [15] for a detailed exposition. A formal proof obligation is presented below.

The implementation (i.e., procedure *apply*) is called *wait-free* if it does not contain operations that can be blocked and if there is a number N such that every call $apply(P, u, z)$ terminates after at most N elementary actions of process P, independently of concurrent calls of *apply* by other processes.

In order to define linearizability as a concrete proof obligation we proceed as follows. When the abstract object has a certain state x, a new invocation u, say by process Q, may lead to a new state y and a result z. We therefore represent the history of the object by a list σ of quadruples $\langle Q, u, y, z\rangle$ with the interpretation that u was an invocation value of process Q that induced the new object state y and the result z. The most recent quadruple is positioned at the head of the list.

We write V to denote the set of such quadruples and V^* to denote the set of finite lists over V. Similarly, U^* and Z^* are the sets of finite lists of elements of U and Z, respectively. Let ε be the empty list. For list σ and element v, let $(v:\sigma)$ be the list obtained by prefixing σ with the singleton v. We define function $stt \in V^* \to X$ that yields the most recent object state, by

$$stt.\varepsilon = x_0$$

$$stt.(\langle Q, u, y, z\rangle:\sigma) = y \tag{3}$$

A list $\sigma \in V^*$ is said to be *acceptable* if and only if σ corresponds to a legal sequential history of the object. This is formalized in predicate *acc*, defined by

$$acc.\varepsilon \equiv \text{true}$$

$$acc.(\langle Q, u, y, z\rangle:\sigma) \equiv \langle stt.\sigma, u, y, z\rangle \in R \wedge acc.\sigma \tag{4}$$

In order to relate the history σ of the object to the actions of a particular process Q, we define $\sigma|Q$ to be the sublist of σ with elements that have first component Q:

$$\varepsilon\,|\,Q = \varepsilon$$

$$(\langle Q, u, y, z\rangle:\sigma)\,|\,Q = \langle Q, u, y, z\rangle:(\sigma\,|\,Q)$$

$$(\langle P, u, y, z\rangle:\sigma)\,|\,Q = (\sigma\,|\,Q) \quad \text{if } P \neq Q \tag{5}$$

We define the functions $in \in V^* \to U^*$ and $out \in V^* \to Z^*$ as the projection functions to the components in U and Z, respectively. So we have

$$in.\varepsilon = \varepsilon$$

$$in.(\langle Q, u, y, z\rangle:\sigma) = u:in.\sigma$$

$$out.\varepsilon = \varepsilon$$

$$out.(\langle Q, u, y, z\rangle:\sigma) = z:out.\sigma \tag{6}$$

So $in.\sigma$ is the list of invocation values of σ and $out.\sigma$ is the list of result values. It follows that $in.(\sigma\,|\,Q)$ and $out.(\sigma\,|\,Q)$ are the sequences of invocation values and result values of process Q as recorded in history σ.

For every process Q, we introduce ghost variables $\beta.Q$ of type U^* and $\gamma.Q$ of type Z^* for the actual lists of invocation values and result values of process Q, both ordered with the most recent value at the head of the list. This means that initially $\beta.Q = \varepsilon$ and $\gamma.Q = \varepsilon$, and that the first action of *apply* is extended with the assignment $\beta.Q := (u:\beta.Q)$ and that the last action of *apply* is extended with $\gamma.Q := (z:\gamma.Q)$.

Definition. The implementation of the data object is called *linearizable* if one can construct a ghost variable σ of type V^*, initially $\sigma = \varepsilon$, that for every execution satisfies the invariants

(CL0) $acc.\sigma$
(CL1) if process Q is not invoking then $\beta.Q = in.(\sigma\,|\,Q)$ and $\gamma.Q = out.(\sigma\,|\,Q)$.

In fact, these conditions imply that σ is a linearization of the invocations according to an order of treatment by the object.

Remark. In [16], we used a different proof obligation which, for deterministic objects, is equivalent to the present one. The present proof obligation is better, for two reasons: it is more symmetric and it is also applicable to nondeterministic objects.

Our implementation satisfies the additional condition that σ only grows:

(CL2) if $\sigma = \sigma_1$ then σ_1 remains a tail of list σ.

This condition, however, is not necessary. Indeed, it is not satisfied by the implementation of [1]. (End of remark)

The concurrent implementation is to be based on a given local implementation

locapply(**in** $u:U$; **var** $y:X$; **out** $z:Z$)

which establishes $\langle w, u, y, z\rangle \in R$ if w is the initial value of y. Each process can use *locapply* on its own private variables (moreover, it is allowed either that u stands for a shared variable to be read, or that z stands for a shared variable to be written).

The other building blocks of the implementation are private variables of arbitrary types, atomic read-write registers for booleans and bounded integers, and *safe shared variables* of arbitrary types. A shared variable is called *safe* if concurrent modifying write operations *of the same value* do not interfere with each other; moreover, as expected, any read operation not concurrent with a modifying write operation must obtain the most recently written value.

A safe variable stores in general a composite value that is not read or written all at once. So there is no assumption of atomicity here. In particular, if two processes are writing different values into the same variable, the final state is undefined, and if one process is reading the variable while another process is modifying it, the value obtained by reading is undefined. Note however that a non-modifying write operation does not interfere with a concurrent read operation.

If a variable is only assumed to be safe, the programmer has to ensure that harmful interferences do not occur. This requirement is formalized by the concept of safe treatment: we define a concurrent program to *treat* a shared variable v *safely* if it satisfies the following two conditions:

(RW-safe) whenever the next action of some process is to read v and the next action of another process is to write $\mathrm{v} := y$, then $\mathrm{v} = y$ holds already.

(WW-safe) whenever different processes have as next actions write actions $\mathrm{v} := y$ and $\mathrm{v} := z$, then $y = z$ holds.

The assertion that the implementation is *wait-free* is proved by constructing an integral state function gw (for gamma weight) that is proportional to the length of list $\gamma.Q$ (up to a bounded error term), that is incremented at every action in *apply* of process Q and that is not decremented by actions of other processes. It will follow that, if process Q performs sufficiently many actions, it obtains a new result (i.e., the implementation is wait-free). More formally stated, the proof obligation is:

(WF) for every process Q, there exist a constant N and a state function vf with $0 < vf \leqq N$ such that the state function

$$gw = N \times (\#\gamma.Q) + vf \tag{7}$$

satisfies, for every action A of process Q and for every action B of other processes, and for all numbers t, the Hoare triples

$$\{gw = t\} \quad A \quad \{gw \geqq t + 1\}$$

$$\{gw = t\} \quad B \quad \{gw \geqq t\} \tag{8}$$

We now prove that condition (WF) implies that the number of actions of process Q between an invocation and the corresponding result is bounded by N. In fact, suppose that Q is idle and has $\#\gamma.Q = r$. At that moment $gw > N \times r$ by formula (7). Now Q submits a new invocation and performs N actions. The Hoare triples (8) then imply that $gw > N \times r + N$ and hence $\#\gamma.Q > r$, again by (7). This proves that, after N actions of Q, process Q has obtained a new result.

Remark. One may propose to replace (WF) by the weaker proof obligation

(AL) there exist a constant $N > 0$ and a state function $gw \leqq N \times (\#\gamma.Q + 1)$ that satisfy the Hoare triples (8).

Condition (AL) expresses that the amortized complexity for invocations of Q is linear, but it allows unboundedly many actions between an invocation of Q and the corresponding result. Indeed, the above argument now fails. We therefore prefer to prove (WF). (End of remark)

To summarize, the list of proof obligations consists of the linearizability conditions (CL0), (CL1), the safety conditions (RW-safe) and (WW-safe) for the non-atomic shared variables, and progress condition (WF).

6 Implementing an arbitrary object

We now turn to the presentation of the wait-free concurrent implementation of an arbitrary data object $\langle X, U, Z, x_0, R\rangle$. The final algorithm will be a variation of the algorithm of [16]. At the end of this section we indicate the changes we have made and the reasons for these changes. In the presentation we give reasons for some design decisions, but we do not give proofs. In fact, in general, when we signal some problem and present some solution to the problem, it will not be apparent that the solution indeed solves the problem.

In the first approximation, *apply0* of Fig. 1, the shared variable sta holds the current state of the object. For each process P, there are variables inv.P and res.P to hold the most recent invocation and result values, and a variable wa.P to indicate that process P has a pending invocation (wa is short for waiting). In the body of the loop, process P treats the invocation of some waiting process Q.

The implementation *apply0* is indeed linearizable. Since the loop body is one big atomic action, however, the grain of atomicity of *apply0* is too coarse. Moreover, *apply0* is not wait-free. In a later refinement, the choice of Q will be implemented in such a way that the algorithm becomes wait-free. At this point, the obvious implementation of the choice of Q would be $Q := P$. This implementation is too eager, however, for then the algorithm cannot be made wait-free when the big atomic action is being split into smaller ones. So we use the variable Q for greater flexibility.

Since we want to split the big atomic action, we prepare the possibility that some process is treating an invocation that was treated before by some other process, or is using an outdated version of the state of the object. We therefore introduce a memory space, called *address*, such that each address can hold an invocation, a state, a result, and a flag to indicate waiting. We introduce a shared variable gg for the address of the current state and we give each process P a variable aa.P for the address of its current invocation. We thus get the declarations

```
type  process = 0..n − 1 {n is number of processes}
;     address = 0..top − 1 {top to be chosen later};
var   gg:  address
;     sta: array address of X
;     inv: array address of U
;     res: array address of Z
;     wa:  array address of boolean
;     aa:  array process of address
```

The types *process* and *address* are subranges of the integers. Without changing the grain of atomicity the program becomes as given in Fig. 2. The program uses two private variables: i for the address of some waiting invocation and y for the state of the object.

```
proc apply 0 (in P: process, u: U; out z: Z);
  [[  inv.P := u
   ;  wa.P := true
   ;  while wa.P do
         ⟨  choose a process Q such that wa.Q
          ;  locapply (inv.Q, sta, res.Q)
          ;  wa.Q := false⟩ od
      z := res.P  ]].
```

Fig. 1. The first approximation

```
proc apply 1 (in P: process, u: U; out z: Z);
  [[  choose a free address aa.P
   ;  inv.(aa.P) := u
   ;  wa.(aa.P) := true
   ;  while wa.(aa.P) do
         ⟨  choose an address i such that wa.i
          ;  y := sta.gg
          ;  locapply (inv.i, y, res.i)
          ;  sta.i := y
          ;  wa.i := false
          ;  gg := i  ⟩od
   ;  z := res.(aa.P)  ]].
```

Fig. 2. The second approximation

```
proc apply (in P: process, u: U; out z: Z);
  [[  if $\#ss.P \geq m$ then A fi
   ;  choose $aa.P \in addr.P \setminus ss.P$
   ;  inv.(aa.P) := u
   ;  nx.(aa.P) := 0
   ;  wa.(aa.P) := true
   ;  $ss.P := ss.P \cup \{aa.P\}$
   ;  while wa.(aa.P) do
           h := gg; bb.P := h
       ;   if h = gg then D fi od
   ;  z := res.(aa.P)  ]].

A: [[ $ss.P := \{gg, aa.P\} \cap addr.P$
  ; for each $T \in process$ with $T \neq P$ do
         i := bb.T
     ;   $ss.P := ss.P \cup (\{i, nx.i\} \cap addr.P)$  od ]].

D: if wa.(aa.P) then
      pf := seq.h; i := aa.pf
    ; if ¬wa.i then i := aa.P fi
    ; ⟨if nx.h = 0 then nx.h := i fi  ⟩
    ; i := nx.h; y := sta.h
    ; locapply (inv.i, y, z)
    ; sta.i := y; res.i := z
    ; seq.i := (pf + 1) mod n
    ; wa.i := false
    ; ⟨ if gg = h then gg := i fi ⟩  fi.
```

Fig. 3. The program as a procedure with refinements

We now turn to the splitting of the big atomic command. This will lead to the program given in Fig. 3.

The first problem to address is the choice of address i as the successor of the current address gg. This choice must be separated from the call of procedure *locapply* and it must be made atomically. We therefore declare

var nx: **array** *address* **of** *address*

and we use $\text{nx}.gg \neq 0$ to record that nx.gg has been chosen as successor of gg. The processes have to achieve consensus concerning the successor of gg. Therefore, nx.gg is made into a consensus register, to be modified by consensus actions of the form

⟨**if** nx.$h = 0$ **then** nx.$h := i$ **fi**⟩

where h is a private copy of the shared variable gg. Notice that address 0 is being used as a nilpointer.

In order to make a fair choice of an address i with wa.i, the current state is tagged with a sequence number seq.gg that indicates the number of the process whose invocation is to be treated next. So we have the additional declaration

var seq: **array** *address* **of** *process*

and the choice of address i is implemented by

```
[[  h := gg
 ;  pf := seq.h   {default process}
 ;  i := aa.pf   {its invocation address}
 ;  if ¬wa.i then i := aa.P fi ]]
```

where the conditional statement gives an eager alternative in the case that the default process is not waiting. Unfortunately, it is not guaranteed that process P itself is still waiting. If P is not waiting anymore, the assignment $i :=$ aa.P in part D of Fig. 3 might lead to a double treatment of P's invocation, i.e. to violation of requirement CL1. This is the reason that in part D the assignment $pf :=$ seq.h is preceded by a second test of wa.(aa.P). It follows that, if process P is not waiting when it enters D, part D is equivalent to skip. It turns out that, if P enters D while waiting but is no longer waiting when $i :=$ aa.P and nx.$h := i$ are executed, there will be invariants that preclude harmful behaviour.

Now the development of part D of Fig. 3 can be concluded. Since it is not guaranteed that process P itself has executed nx.$h := i$, the conditional statement is followed by $i :=$ nx.h to obtain the predicate nx.$h = i$ (see invariant kp8 in part 6 of Sect. 13 below). A private variable z is introduced, so that the call of *locapply* only refers to one shared variable (inv). The sequence number seq.i of the new state is obtained from the old one by circular incrementation. Finally, variable gg, the pointer to the current state, is modified only if it still equals the value of its private copy h. Otherwise, the work done in part D was superfluous (an important aspect of the proof is to show that all superfluous work is harmless; this follows, however, when the proof obligations mentioned at the end of Sect. 5 are met).

It remains to implement the choice of a free address for aa.P. In order to avoid that different processes choose the same free address concurrently, we give each process its own pool of addresses. The range *address* is therefore partitioned into sets *addr*.Q given by

$$k \in addr.Q \equiv m \times Q < k \leqq m \times (Q+1) \qquad (9)$$

where m is sufficiently large. We take $top = m \times n + 1$. It follows that every nonzero address belongs to precisely one process. Slightly deviating from [16], we give each process Q a global private variable $ss.Q$ to stand for the set of addresses in its pool that are not free for a new invocation. So we have the additional declaration

var *ss*: **array** *process* **of set of** *address*

The variables $ss.Q$ is only accessed by process Q. Since it is global, it need not be computed in every invocation.

When process P has no free addresses left (i.e., when $\# ss.P \geqq m$), it executes the garbage collector command A of Fig. 3. Command A reconstructs $ss.P$ as the set of addresses in the pool of P which are still referred to by other processes T. For this purpose, command A needs the values of the private variables h. These values are broadcasted by a shared variable

var bb: **array** *process* **of** *address*

To get accurate broadcasting, an additional test is placed in the loop body. This body therefore becomes

```
⟦  h:= gg; bb.P:= h
;  if h = gg then D fi ⟧
```

The two-seemingly superfluous consecutive tests $h = gg$ and wa.(aa.P) cannot be omitted. The program even becomes incorrect if the tests are permuted. The danger to avoid here is that process P may be idle for an extended period (e.g. just before the assignment bb.$P := h$) and then start executing again (this was described once by M.O. Rabin as the problem of the senile politician). The incorrectness in case of permutation of the two tests requires a delicate scenario, which is given in Sect. 7 of [16].

It follows from (9) that $addr.P$ has m elements. Since fragment A clearly establishes the postcondition $\# ss.P \leqq 2 \times n$, the subsequent choice of aa.P is possible if $2 \times n < m$. Thus, at this point, we can state the requirements on the system parameters n, m and top:

$$n > 0 \wedge 2 \times n < m \wedge top = m \times n + 1 \qquad (10)$$

The choice of aa.P in $addr.P \backslash ss.P$ can be implemented in a completely arbitrary way. The obvious way is to use linear search. Since it is a private computation, it need not concern us here, but can be dealt with by the implementer.

In this way, we arrive at procedure *apply* as given in Fig. 3. It remains to say that nx.(aa.P) := 0 is a necessary reset operation and that $ss.P := ss.P \cup \{$aa.$P\}$ records that address aa.P is put into usage. The first assignment to $ss.P$ in part A is motivated by the fact that at that part of the program we have nx.gg = aa.P if nx.gg $\in addr.P$, and by the condition that an atomic command may contain at most one variable that can be modified by other processes (recall that we only give design considerations and that the proof is postponed).

Crucial atomic commands in this algorithm are the two commands enclosed in $\langle\rangle$ in fragment D. The atomic conditional modification of nx.h indicates that nx is an array of consensus registers, see (1). In view of its atomic conditional modification, variable gg is a compare&swap register, see (2).

By now the data structure is complete. It is subject to the following initial condition:

$$\begin{aligned} & mem.gg \wedge \text{sta.gg} = x_0 \wedge \text{nx.gg} = 0 \wedge \text{seq.gg} = 0 \\ & \wedge (\forall k \in address :: \neg \text{wa}.k) \\ & \wedge (\forall T \in process :: mem.(\text{aa}.T) \wedge \{\text{gg}\} \cap addr.T \subseteq ss.T) \end{aligned} \qquad (11)$$

where $mem.k$ expresses $0 < k < top$.

The initial value 0 of seq.gg is not required in [16], but 0 is a natural starting point and this choice was more convenient for the mechanical proof. In [16], the initial conditions on aa.T and $ss.T$ are much stronger. The present initial conditions are made possible by some modifications in the invariants.

The algorithm of Fig. 3 only deviates from [16] in the following points. Instead of $ss.P$, we used in [16] the equivalent variable $s.P$, related to $ss.P$ by means of

$$s.P = ss.P \cup (address \backslash addr.P)$$

$$ss.P = s.P \cap addr.P \qquad (12)$$

The treatment of $ss.P$ is the direct translation of the treatment of $s.P$ in [16]. There is one point where a second program transformation has been applied. In the first conditional command of *apply*, the translated test would have been $ss.P = addr.P$. This has been replaced by $\# ss.P \geqq m$ to avoid the necessity of a test on set equality. The present version is also more convenient for the prover.

The local variables pf, y and z have been introduced here to eliminate commands in which two or more shared variables are accessed. For example, part D as developed in [16] contains composite assignments $i :=$ aa.(seq.h), and seq.$i :=$ (seq.h + 1) **mod** n. The present program uses pf to keep the value of seq.h. Of course, such a separation of variable accesses requires the introduction of new invariants.

The last point is that, in [16], we used the atomicity rule ([2] Theorem 6.26) to argue that the assignments to wa.(aa.P) and $ss.P$ in *apply* could be regarded as being combined in a single atomic command. For the prover it turned out to be only a minor complication to treat the two assignments as separate commands. See the end of part 6 in Sect. 13.

7 A nonterminating goto program

In this section, the program is reformulated to facilitate the proof. We add the history variables required by (CL0) and (CL1) and we add labels to discuss the execution of the processes. We eliminate composite constructs for **if** and **while** by means of **goto** commands. The resulting program is given in Fig. 4. We now discuss the transformations applied in more detail.

For every process Q, at every moment, at most one incarnation of procedure *apply* is active. We may therefore replace procedure *apply* by one unbounded repetition for every process, with the body of *apply* as body of the loop. The input-parameter u of *apply* may be regarded as a value that is chosen non-deterministically each time that Q enters its loop body.

We use the auxiliary variables β, γ and σ, as introduced in Sect. 5. This means that, at every choice of a new invocation value u, the list $\beta.P$ is replaced by $(u : \beta.P)$. At the end of the loop body, list $\gamma.P$ is replaced by (res.(aa.P): $\gamma.P$). Now the result parameter z can be omitted, but we must remember that res.(aa.P) is actually being read.

We choose to update σ in the atomic command in fragment D where wa.i is made false. More precisely, if

```
     0    choose $u$; $\beta.P := u : \beta.P$
     1    if $\#ss.P < m$ goto 8
     2 A:   $ss.P := \{gg, aa.P\} \cap addr.P$
     3      $plist := process \setminus \{P\}$
     4      if $plist = \varepsilon$ goto 8
     5        shuffle $plist$; $i :=$ bb.(car plist)
     6        $ss.P := ss.P \cup (\{i, nx.i\} \cap addr.P)$
     7        $plist :=$ (cdr plist); goto 4
     8    choose $aa.P \in addr.P \setminus ss.P$
♡    9    $inv.(aa.P) := u$
    10    $nx.(aa.P) := 0$
    11    $wa.(aa.P) :=$ true
    12    $ss.P := ss.P \cup \{aa.P\}$
    13    if $\neg wa.(aa.P)$ goto 31
    14      $h := gg$
    15      $bb.P := h$
    16      if $h \neq gg$ goto 13
    17    D:  if $\neg wa.(aa.P)$ goto 13
    18        $pf := seq.h$
    19        $i := aa.pf$
    20        if $wa.i$ goto 22
    21          $i := aa.P$
    22        if $nx.h = 0$ then $nx.h := i$ fi
    23        $i := nx.h$
♡   24        $y := sta.h$
♡   25        locapply ($inv.i, y, z$)
♡   26        $sta.i := y$
♡   27        $res.i := z$
    28        $seq.i := (pf + 1)$ mod $n$
    29        if $wa.i$ then $\sigma := \langle pown.i, inv.i, y, z\rangle : \sigma$ fi
              ;  $wa.i :=$ false
    30        if $gg = h$ then $gg := i$ fi ; goto 13
♡   31    $\gamma.P := res.(aa.P) : \gamma.P$ ; goto 0
```

Fig. 4. The program as a **goto** program

wa.i holds, this command is extended with

$\sigma := \langle pown.i, \text{inv}.i, y, z\rangle : \sigma$

where function *pown* gives the process that submitted the invocation located at address i, i.e.,

$$pown.i = Q \quad \text{if } i \in addr.Q \tag{13}$$

The for-loop in fragment A of Fig. 3 involves the nondeterminate choice of a process T that is to be treated next. In order to model this nondeterminate choice, we have chosen to introduce a local variable *plist* of process P such that $T \in plist$ means that T has yet to be treated in the for-loop. The selection of an arbitrary element of *plist* is implemented by a nondeterminate shuffle of the list followed by selection of its *car* (i.e., head).

In the mechanical proof, we need the program counters of the processes to keep track of the concurrent executions. Having introduced the program counters, we eliminate the loops and the composite conditionals by means of goto commands. In this way we arrive at the program for process P given in Fig. 4. In commands 5 and 7, we use the LISP symbols *car* and *cdr* to denote the head and the tail of list *plist*.

Now that the program of Fig. 4 has been concluded, we can discuss the atomicity of the commands. The commands 0, 1, 3, 4 and 7 only refer to private variables and may therefore be regarded as atomic. The variables wa.k, seq.k ($k \in address$) and aa.P, bb.P ($P \in process$) are atomic read-write variables. An action on such a variable may be combined atomically with actions on private variables. We may therefore assume atomicity of the commands 5, 8, 12, 15, 18, 19, 20, 21, 28. Since σ is only an auxiliary variable, the conditional assignment to σ and the inspection of inv.i can be combined atomically with the assignment to wa in 29.

Since gg is a compare and swap variable and nx.h is a consensus variable, the commands 6, 14, 16, 22, 23, 30 are regarded as atomic. Notice that actions on these variables may also be combined atomically with actions on private variables. Since aa.P is only modified by process P itself, commands 2, 10, 11, 13 and 17 may also be regarded as atomic.

The remaining commands 9, 24, 25, 26, 27 and 31 are not atomic. They are the central actions of value transfer and computation. These commands are marked with the symbol ♡. Commands 24, 25 and 31 are read actions of shared variables sta.k, inv.k and res.k. Commands 9, 26 and 27 are write actions for such variables. These variables are assumed to be *safe* variables, as discussed in Sect. 5.

8 Modelling the program in NQTHM

In the next sections, we describe how the program of Fig. 4 is treated with the theorem prover NQTHM. We go into the NQTHM details for two reasons. On the one hand, we want to give readers who want to use NQTHM for similar purposes an impression of the kind of hurdles one encounters. On the other hand, we want to give the insiders of NQTHM a handle to certify our claims.

Many proofs published by Boyer, Moore, and their associates (eg. [22]) rephrase definitions and lemmas into more tranditional notation to make it more broadly accessible. We have three reasons for not doing so. Firstly, we want to give a faithful representation of the encodings used for readers who consider the use of NQTHM for a similar project. Secondly, it is our experience, e.g. while reading [22] with a group of non-users of NQTHM, that rendering into more traditional notation frequently raises questions that can only be solved by discussion of the NQTHM notation. Thirdly, this paper uses the interpretation of quoted constants for which a more traditional notation is not readily available.

We refer to the Handbook [8] for the description of the language, the logic and the theorem-proving capabilities of NQTHM. The language of NQTHM is a variation of pure LISP. In particular, it is an untyped, purely functional language in which almost all functions are total.

We treat the program as a syntactic constant and use it to construct a function step that yields the new global program state in terms of the old program state x, the acting process p and a nondeterminate argument nd.

The first thing to do is to make a formalism to structure the state space of the program that is to be interpreted by means of a declaration of the program variables. As a preparation for the concrete declaration, we then introduce (in Fig. 5) NQTHM constants (n), (m) and (top) to represent the constants n, m and *top* as restricted by Formula (10). In fact, the constrain expression introduces two function symbols n and m of arity zero and postulates (n) > 0 and (m) $>$ (n) $\times 2$. In the NQTHM logic, this implies that (n) and (m) are natural numbers. The term

```
(constrain axiom-constants (rewrite)
  (and (lessp 0 (n))                     ; 0 < (n)
       (lessp (times (n) 2) (m)) )       ; (n) * 2 < (m)
  ((n (lambda () 1)) (m (lambda () 3))) )

(defn top () (addi (times (n) (m))) ) ; (top) = 1 + (n)*(m)
```

Fig. 5. Declaration of constants

```
(defn declaration ()
  (list '(sigma) '(gg) (list 'wa (top)) (list 'nx (top))
    (list 'inv (top))  (list 'res (top)) (list 'sta (top))
    (list 'seq (top))  (list 'aa (n))    (list 'bb (n))
    (n) '(pc) '(uu) '(beta) '(gamma) '(ss) '(hh)
      '(ii) '(plist) '(pf) '(yy) '(zz) ) )
```

Fig. 6. Declaration of variables and arrays

(rewrite) indicates to the prover how the axiom is to be used; axiom-constants is the name of the axiom. The last line provides a model ((n) = 1 and (m) = 3) that is used by NQTHM to verify the consistency of the axiom. The second expression defines the constant (top). Semicolons with subsequent characters are regarded as comment.

The concrete declaration of the program state is introduced by Fig. 6. It declares shared variables 'sigma (for σ) and 'gg and shared arrays 'wa up to 'bb together with the lengths of these arrays. It next declares an anonymous array of private states of length (n). Each private state consists of private variables 'pc up to 'zz. We have chosen to represent the private variables u, h, i, y, z by 'uu, 'hh, etc. The reader should not be worried by the parentheses and the quotes in Fig. 6. Explanation of those details would require a genuine introduction to NQTHM and an explicit description of our data representation. Let it suffice to say that the quotes are needed since the names in the program must be represented as values.

Since the proof has to argue about them, the auxiliary variables 'sigma, 'beta and 'gamma are treated as ordinary variables. The private variable 'pc is the program counter of its process.

The declaration serves to structure the program state x as a tree. The associated formalism contains a function val such that (val dec u x) yields the subtree of tree x according to root path u in declaration dec. For example, corresponding to the interpreted variable 'wa, we define function wa by

```
(defn wa (k x)
  (val (declaration) (list 'wa k) x) )
```

Now (wa k x) is the value wa.k in state x.

Having structured the program state we turn to the program. The data object to be linearized is unknown. Since it is supposed to be total and deterministic, it is modelled by two unknown functions newst and newres that determines the new state of the object and the new result with as arguments the old state and the new invocation value. Similarly, the initial state x_0 is modelled by an unknown function initobj with no arguments. This is done by the declarations in

```
(dcl initobj ())     ; the initial state of the object
(dcl newst (u x))    ; new state for invocation u and old
                       state x
(dcl newres (u x)) ; the new result
```

The program is submitted to the prover as the syntactic entity given in Fig. 7. It is an association list with a modification command for every value of the program counter. This modification command is a list of assignments, to be executed sequentially. The goto commands in 1, 7, 13, etc. now have become assignments to 'pc. Compare Fig. 4.

A small general purpose interpreter step is provided, see Fig. 8. It is built by means of a function modify* which is such that (modify* dec p nd al x) is the new state built from the old state x by means of assignment list al, the arguments p and nd, and the declaration dec. The interpreter step presupposes that 'pc is a private variable of the processes. By default it increments 'pc prior to execution of the command. The parameter p gives the identity of the acting process; it gives values to the syntactic term (self) in

```
(defn program ()
 ' ( (0 .  ( ((uu) . (extern))
             ((beta) . ( (extern) . (beta) )) ))
    (1 .  ( ((pc) . (if (lessp (length (ss)) (m)) 8 2)) ))
    (2 .  ( ((ss) . (ladd2 (gg) (aa (self)) (self) (m) ())) ))
    (3 .  ( ((plist) . (delete (self) (segment (n)))) ))
    (4 .  ( ((pc) . (if (listp (plist)) 5 8)) ))
    (5 .  ( ((plist) . (shuffle (extern) (plist)))
             ((ii) . (bb (car (plist)))) ))
    (6 .  ( ((ss) . (ladd2 (ii) (nx (ii)) (self) (m) (ss))) ))
    (7 .  ( ((plist) . (cdr (plist)))
             ((pc) . 4) ))
    (8 .  ( ((aa (self)) . (choose (self) (ss) (extern))) ))
    (9 .  ( ((inv (aa (self))) . (uu)) ))
    (10 . ( ((nx (aa (self))) . 0) ))
    (11 . ( ((wa (aa (self))) . (true)) ))
    (12 . ( ((ss) . (add-to-set (aa (self)) (ss))) ))
    (13 . ( ((pc) . (if (wa (aa (self))) 14 31)) ))
    (14 . ( ((hh) . (gg)) ))
    (15 . ( ((bb (self)) . (hh)) ))
    (16 . ( ((pc) . (if (equal (hh) (gg)) 17 13)) ))
    (17 . ( ((pc) . (if (wa (aa (self))) 18 13)) ))
    (18 . ( ((pf) . (seq (hh))) ))
    (19 . ( ((ii) . (aa (pf))) ))
    (20 . ( ((pc) . (if (wa (ii)) 22 21)) ))
    (21 . ( ((ii) . (aa (self))) ))
    (22 . ( ((nx (hh)) . (if (zerop (nx (hh)))
                                              (ii) (nx (hh)))) ))
    (23 . ( ((ii) . (nx (hh))) ))
    (24 . ( ((yy) . (sta (hh))) ))
    (25 . ( ((zz) . (newres (inv (ii)) (yy))) ; sequential!
              ((yy) . (newst (inv (ii)) (yy))) ))
    (26 . ( ((sta (ii)) . (yy)) ))
    (27 . ( ((res (ii)) . (zz)) ))
    (28 . ( ((seq (ii)) . (sucmod (pf) (n))) ))
    (29 . ( ((sigma) . (if (wa (ii))
                            (((pown (ii) (m)) (inv (ii))
                                               (yy) (zz)). (sigma) )
                            (sigma) ))
             ((wa (ii)) . (false)) ))
    (30 . ( ((gg) . (if (equal (gg) (hh)) (ii) (gg)))
             ((pc) . 13) ))
    (31 . ( ((gamma) . ( (res (aa (self))) . (gamma)))
             ((pc) . 0) )) ) )
```

Fig. 7. The program to be interpreted

```
(defn step (p nd x)
  (if (and (lessnum p (n))
           (assoc (pc p x) (program)) )
    (modify* (declaration) p nd
             (cdr (assoc (pc p x) (program)))
             (addlpc p (pc p x) x) )
    x ) )
```

Fig. 8. The interpreter of declaration and program

```
(defn pown (k m)
  (if (zerop k) nil (quotient (sub1 k) m) ))
  ; if k = 0 then "nil" else (k - 1) div m fi

(defn sucmod (q n)
  (if (lessp (add1 q) n) (add1 q) 0) )
  ; if q + 1 < n then q + 1 else 0 fi
```

Fig. 9. The definitions of pown and sucmod

```
(constrain shuffle-axiom (rewrite)
  (and (equal (length (shuffle nd s)) (length s))
       (equal (listp (shuffle nd s)) (listp s))
       (equal (member y (shuffle nd s)) (member y s)) )
  ((shuffle (lambda (nd s) s))) )

(constrain choose-axiom (rewrite)
  (implies (and (numberp p)
                (lessp (length s) (m)) )
           (and (equal (pown (choose p s nd) (m))
                       p)
                (not (member (choose p s nd) s)) ) )
  ((choose (lambda (p s nd) (ownchoose p s)))) )
```

Fig. 10. The axiomatic introduction of shuffle and choose

the program. The parameter nd is used to give values to the syntactic term (extern) that governs the nondeterminacy in the commands 0, 5 and 8. Notice that the program refers in a direct way to the private variables of the acting process: the interpreter does not allow reference to private variables of other processes.

The program contains the standard NQTHM functions if, true, false, lessp, car, cdr, listp, equal, zerop. It contains the defined functions length, ladd2, delete, segment, sucmod, pown. These functions are straightforward translations of the expressions used in Fig. 4, see lines 1, 2, 3, 3, 28 and 29. The definitions of pown and sucmod are shown in Fig. 9.

The functions shuffle and choose are introduced axiomatically by means of the expressions in Fig. 10. We give shuffle and choose an additional argument nd to model the nondeterminacy supposed in the program of Fig. 4. In fact, we shall make no assumption about nd. So at every step its value can be different. It is easy to provide a model for shuffle. The construction of function ownchoose as a model for choose turned out to be nontrivial.

The initial condition of the state is characterized by a predicate initpred that combines the initial condition (11) and the initialization of the program counters and the auxiliary variables

$$\sigma = \varepsilon \wedge (\forall T \in process :: pc.T = 0 \wedge \beta.T = \varepsilon \wedge \gamma.T = \varepsilon)$$

We have taken satisfiability of initpred to be our first proof obligation. We shall not discuss the proof here.

9 The main theorems to be proved

In this section, we present the theorems that comprise the main proof obligations. These theorems have all been proved by NQTHM, but there is a large gap between the program as described above and these theorems. The way we have filled that gap is described later.

In distributed programming there is no complete agreement concerning the concept of invariant. We take the following definitions. A predicate J is called a *strong invariant* if it holds initially and is preserved by every possible step, i.e., if it satisfies the two implications

$$(\text{initpred x}) \Rightarrow (J\ \text{x}),$$
$$(J\ \text{x}) \Rightarrow (J\ (\text{step p nd x}))$$

for all values x, p and nd. A predicate J is called an *invariant* if it is implied by a strong invariant.

Proof obligation (CL0) is fulfilled by the construction of a strong invariant j-object that satisfies the lemmas of Fig. 11. In fact, predicate acc is the direct translation of *acc* as defined in (4). The first lemma states that j-object implies *acc*.σ and the other two lemmas state that j-object is a strong invariant.

For proof obligation (CL1), we translate the definitions (5) and (6) to construct functions linvlist and lreslist such that

$$(\text{linvlist q x}) = in.(\sigma \mid \text{q}) \text{ in state x}$$
$$(\text{lreslist q x}) = out.(\sigma \mid \text{q}) \text{ in state x}$$

Then proof obligation (CL1) is fulfilled by the lemmas in Fig. 12. Here, j-invoc and j-result are strong invariants when applied to a genuine process q. We have suppressed the lemmas for the strong invariance of j-result. Process q is idle if and only if its program counter is zero. Therefore the two final lemmas of Fig. 12 imply (CL1).

Proof obligation (WF) of Sect. 5 is fulfilled by the lemmas in Fig. 13. We first give the definition of function gw that corresponds to (7). So we take $N =$ (c22) and vf = (c22) − (vfloop). For the present purposes the value of (c22) is irrelevant. We then prove that predicate j-progress, which is a strong invariant, implies the bounds required for function vfloop. The two final lemmas of Fig. 13 are the Hoare triples (8). The functions lessp and leq are prefix functions for $<$ and $\leqq$, respectively.

The proof obligations that the shared variable sta is treated safely, are fulfilled in Fig. 14. The first lemma states that variable 'sta is being read only by command 24, at index 'hh, and that 'sta is written only in command 26, at index 'ii with the value of 'yy. The other two lemmas verify RW-safety and WW-safety of 'sta. We use the strong invariant globinvariant, which is a conjunction of universal quantifications of a large number of invariants. The

```
(prove-lemma j-object-implies-acc ()
  (implies (j-object x) (acc (sigma x))) )

(prove-lemma init-j-object (rewrite)
  (implies (initpred x) (j-object x)) )

(prove-lemma j-object-kept-valid (rewrite)
  (implies (j-object x) (j-object (step p nd x))) )
```

Fig. 11. Proof obligation (CL0)

```
(prove-lemma init-j-invoc (rewrite)
  (implies (and (initpred x)
                (lessnum q (n)) )
           (j-invoc q x) ) )

(prove-lemma j-invoc-kept-valid (rewrite)
  (implies (j-invoc q x)
           (j-invoc q (step p nd x)) ) )

(prove-lemma invoc-correct ()
  (implies (and (j-invoc q x)
                (zerop (pc q x)) )
           (equal (beta q x) (linvlist q x)) ) )

(prove-lemma result-correct ()
  (implies (and (j-result q x)
                (zerop (pc q x)) )
           (equal (gamma q x) (lreslist q x)) ) )
```

Fig. 12. Proof obligation (CL1)

```
(defn gw (q x) ; gammaweight
  (plus (times (length (gamma q x)) (c22))
        (difference (c22) (vfloop q x)) ) )

(prove-lemma range-vfloop ()
  (implies (j-progress q x)
           (and (not (zerop vfloop q x)))
                (lessp (vfloop q x) (c22)) ) ) )

(prove-lemma gw-increases-eq ()
  (implies (j-progress p x)
           (lessp (gw p x)
                  (gw p (step p nd x)) ) ) )

(prove-lemma gw-ascends-dif ()
  (implies (and (j-process q x)
                (not (equal p q)) )
           (leq (gw q x)
                (gw q (step p nd x)) ) ) )
```

Fig. 13. Bounded delay: the proof obligation (WF)

proof obligations for inv and res are handled in the same way.

It may be granted here that this verification of RW-safety and WW-safety is not completely formal. A human interpreter is needed to verify that Fig. 14 indeed captures safe treatment of the variable sta. A completely formal verification would require some careful trading between the syntactic and the semantic level, which a human interpreter of Fig. 14 does almost automatically. We refrained from this formalization, since our main concern in this project was the correctness of the concurrent treatment of shared data and not the formalization and verification of the whole trajectory from specification to implementation.

10 Using NQTHM

From a global point of view, NQTHM has served us as a proof checker rather than a theorem prover. We had to invent all definitions, theorems and intermediate lemmas. In many cases we had to provide one or more hints in order to get a lemma proved.

The most straightforward way of hinting is to advise the prover that it may *use* a specific instantiation of a lemma proved previously. Such use hints are by no means always

```
(prove-lemma sta-touched ()
  (and (equal (areadprog (progam) 'sta)
              '((24 . (((hh))))) )
       (equal (awriteprog (program) 'sta)
              '((26 . (((ii)) . (yy))))) ) ) )

(prove-lemma sta-read-safe ()
  (implies (and (globinvariant x)
                (lessnum p (n))
                (lessnum q (n))
                (equal (pc p x) 26)
                (equal (ii p x) (hh q x))
                (equal (pc q x) 24) )
           (equal (sta (ii p x) x) (yy p x)) ) )

(prove-lemma sta-write-safe ()
  (implies (and (globinvariant x)
                (lessnum p (n))
                (lessnum q (n))
                (equal 26 (pc p x))
                (equal 26 (pc q x))
                (equal (ii p x) (ii q x)) )
           (equal (yy p x) (yy q x)) ) )
```

Fig. 14. Safe treatment of variable sta

necessary or productive. A more subtle way of hinting is to *disable* some definitions or lemmas. For example, if a lemma can be proved by using an already established property of a function, it may be useful to disable the definition of the function. For, otherwise, it may be that the definition is unfolded and that application of the property is not considered. It is possible and often useful to disable a definition or lemma globally when its main corollaries have been obtained. In that case, the prover can be allowed to use the disabled information by means of a hint that *enables* it again.

In order to avoid unnecessary renaming, we always use p to denote the acting process; other processes are q and x. We always use x and nd to denote the old program state and a nondeterminate argument, respectively. This has the effect that the new program state is always (step p nd x).

11 The global structure of the proof

The proof we have constructed consists of 6485 lines of NQTHM code. It contains 1057 events, i.e., definitions, lemmas and disabling commands. It can be divided in eleven parts. Mainly to show the relative sizes of the parts, we give for each of them the number of lines and the number of events.

1. Declaration: the structuring of the state, 545 lines, 105 events.
2. The program, its functions and initial conditions, 578 lines, 104 events.
3. The semantic lemmas, 864 lines, 182 events.
4. Accumulating easy invariants, 356 lines, 67 events.
5. The top-level invariants, 525 lines, 76 events.
6. Invariants of the pointer structure, 1217 lines, 152 events.
7. Invariants of value transfer and computation, 368 lines, 49 events.
8. Aggregation of invariants, 373 lines, 84 events.
9. Proof obligations of correctness, 176 lines, 25 events.

10. Predicates for bounded delay, 843 lines, 114 events.
11. Bound functions and bounded delay, 640 lines, 99 events.

12 Initial set-up

The parts 1, 2 and 3 form an initial set-up that can serve as a prototype for many distributed algorithms.

Part 1. Declaration: the structuring of the state
The contents of this part of the proof have been described in Sect. 8. In addition to the function val mentioned there, the formalism contains a function put such that (put dec u w x) is a new state built from x by substituting w for the subtree of x of root path u in dec. Now NQTHM is guided to prove lemmas that describe how the value of a variable changes when the state is modified by means of a put command. These low-level lemmas form a preparation for the lemmas to be described in part 3 of the proof.

Part 2. The program, its functions and initial conditions
This part contains the definitions in Figs. 7 and 8 and the definitions of the functions length, ladd2, delete, segment, sucmod and pown that are used in the program of Fig. 7. It also contains the expressions of Fig. 10, the definition of initpred and the proofs of satisfiability of the axioms and of initpred.

Part 3. The semantic lemmas
A list of rewrite lemmas is constructed to describe the effect of every command on every program variable. When this list is completed the interpreter can leave the stage (i.e., be disabled) and the program itself can almost be forgotten. As an example we give in Fig. 15 the new value of wa.k when process P performs one step. This lemma states that wa.k only changes when some process P executes command 11 or 29, and k equals aa.P or $i.P$ (respectively); it also gives the new values: *ture* and *false*.

Remark. This part is rather dull. It may be possible to construct a tool that can generate all the necessary lemmas. The results of this part, however, form a checkpoint of the correctness of the interpretation by the interpreter and of the data representation. (End of remark)

13 Invariants of the program

In the parts 4, 5, 6, 7, 8 and 10 of the proof, we construct lots of invariants. We only give a few of them in this report. Of course, the input file to the prover contains all invariants. Most invariants can be recognized by lemma names of the form J-kept-valid. The list of invariants of [16] is a close approximation, but the finer grain of atomicity here requires some reformulations and some new invariants.

Since the commands have been made as small as possible, a variable is often modified at other points in the program than would be most convenient for the phrasing of the invariants. The difficulty can often be solved by the introduction of a *companion*, i.e., a state function that is usually equal to the variable but changes at other points in the program. We give some examples below, see function *vpl* as discussed below in part 4, and function *vs* in part 6. The concept of companion is not completely formal, but it seems to be very useful in completely formal proofs of algorithms with fine grain concurrency.

```
(prove-lemma wa-new (rewrite)
  (equal (wa k (step p nd x))
         (if (lessnum k (top))
             (if (and (equal 11 (pc p x))
                      (equal (aa p x) k) )
                 (true)
                 (if (and (equal 29 (pc p x))
                          (equal (ii p x) k) )
                     (false)
                     (wa k x) ) )
                 (wa k x) ) ) )
```

Fig. 15. The new value of 'wa

Part 4. Accumulating easy invariants
The invariants of part 4 are obtained incrementally. The first aim is to prove that, in command 8, process P can indeed choose an address aa.P in $addr.P\backslash ss.P$. So we have to show that this set is nonempty. For this purpose, we use the axiomatization of function choose in Fig. 10 together with the invariant ks2 which asserts that $\# ss.Q < m$ when $pc.Q = 8$. It is given by

```
(defn ks2 (q x)
  (implies (equal 8 (pc q x))
           (lessp (length (ss q x)) (m)) ) )
```

The proof of invariance of ks2 is based on Formula (10) together with the invariant ks1 which asserts

$$pc.Q \in \{3, 4, 5, 6, 7\} \Rightarrow \# ss.Q + 2 \times \# vpl.Q \leqq 2 \times n$$

Here, $vpl.Q$ is a companion of the private variable *plist*: i.e., a state function usually equal to *plist* which is modified at more convenient points in the program. More precisely, $vpl.Q = plist.Q$ if $pc.Q \neq 3, 7$, and $vpl.Q = process\backslash\{Q\}$ if $pc.Q = 3$ and $vpl.Q = (cdr\ plist.Q)$ if $pc.Q = 7$. So when $vpl.Q$ differs from $plist.Q$, it has the value that $plist.Q$ will get after the next action of Q. Function *vpl* is introduced to allow the separation of the commands 2 and 3 and of the commands 6 and 7. In fact, by inspection of Fig. 4, one sees that, if *vpl* would be replaced by *plist*, predicate ks2 could become invalid at command 6.

In order to prove the invariance of ks1, we need the obvious invariant ks0 which asserts that $plist.Q \neq \varepsilon$ when $pc.Q \in \{5, 6, 7\}$.

Part 5. The top-level invariants
This part contains a preparation of the lemmas in Figs. 11, 12. Here, predicate j-invoc is defined and the three lemmas about j-invoc are proved. The lemmas about j-object and j-result are only proved under some assumptions, which are justified in the parts 6, 7 and 8. One of the invariants introduced here is jj0, which expresses that the most recent state of the object $stt.\sigma$ equals sta.gg or sta.(nx.gg) and that the second case only occurs if $nx.gg \neq 0$ and $\neg$wa.(gg). Figure 16 shows how this is expressed. Predicate jj0 is one of the constituents of the strong invariant j-object mentioned earlier.

```
(defn ng (x) (nx (gg x) x))

(defn gphase (x)
  (or (zerop (ng x))
      (wa (ng x) x) ) )

(defn jj0 (x)
  (equal (stt (sigma x))
         (if (gphase x)
             (sta (gg x) x)
             (sta (ng x) x) ) ) )
```

Fig. 16. Definition for jj0

```
(defn kg0 (k x)
  (or (zerop k)
      (not (wa k x))
      (equal 0 (nx k x)) ) )

(prove-lemma kg0-kept-valid (rewrite)
  (implies (and (kp0 p x)
                (nwah p x)
                (kg0 k x))
           (kg0 k (step p nd x)) )
  ((do-not-induct t)) ) ; [0.0 1.7 0.3]
```

Fig. 17. The invariant kg0

Part 6. Invariants of the pointer structure

The main difficulty of the program is that all processes concurrently inspect and modify a shared pointer structure. It follows that the proof requires many invariants concerning this pointer structure. We only give some examples. The invariant kg0 of Fig. 17 expresses

$$k \neq 0 \wedge \mathrm{wa}.k \Rightarrow \mathrm{nx}.k = 0,$$

i.e., a waiting invocation has no successor. Lemma kg0-kept-valid announces the invariance of kg0. This lemma uses two other invariants kp0 and nwah. Predicate kp0 expresses that $\mathrm{nx}.(\mathrm{aa}.P) = 0$ when $pc.P = 11$. Predicate nwah expresses that $\mathrm{wa}.(h.P)$ is false when $17 \leqq pc.P < 31$. Both kp0 and nwah are invariant (this is not obvious, since other processes may modify the arrays nx and wa).

The last line of Fig. 17 is a hint to the prover that it should not use induction. In the final proof, this hint is superfluous, but during the design it serves to terminate failing proof attempts. After the comment separator ";", we give the time triple reported by NQTHM: the first number is the number of seconds spent for input of the lemma, the second number the number of seconds spent for the proof, the third number is the number of seconds spent printing information to the user. The numbers given here were obtained on a HP 9000-720. Since the invariance of kg0 depends on the invariance of the other predicates, the actual proof of invariance of kg0 is postponed to part 8 below.

Since we need the invariant kg0 for all addresses k, we also form kkg0, the universal quantification of (kg0 k x) for all values k less than top, see Fig. 18. The invariance of kkg0 is proved by induction. The last line of kkg0-kept-valid indicates to the prover that it should not unfold the definitions of kg0, nwah, kp0. Notice that top is a variable symbol here, whereas (top) is a constant. The lemma will only be used with top = (top). If it would have been stated

kkg0: $(\forall k: 0 < k < \mathrm{top}: \neg \mathrm{wa}.k \vee \mathrm{nx}.k = 0)$

```
(defn kkg0 (top x)
  (if (zerop top) (true)
      (and (kg0 (sub1 top) x)
           (kkg0 (sub1 top) x) ) ) )

(prove-lemma kkg0-kept-valid (rewrite)
  (implies (and (kp0 p x)
                (nwah p x)
                (kkg0 top x))
           (kkg0 top (step p nd x)) )
  ((disable kg0 nwah kp0)) ) ; [0.0 0.1 0.1]
```

Fig. 18. The invariant kkg0, the quantification of kg0

kp8: $24 \leqq \mathrm{pc}.Q < 31 \Rightarrow \mathrm{nx}.(h.Q) = i.Q$

```
(defn kp8 (q x)
  (implies (between 24 31 (pc q x))
           (equal (nh q x) (ii q x)) ) )

(prove-lemma kp8-eq (rewrite)
  (implies (and (lessnum p (n))
                (kp8 p x) )
           (kp8 p (step p nd x)) ) ) ; [0.0 9.9 10.4]

(prove-lemma kp8-dif (rewrite)
  (implies (and (not (equal p q))
                (ahn p q x)
                (memi q x)
                (kp8 q x))
           (kp8 q (step p nd x)) ) ) ; [0.0 2.7 0.6]

(prove-lemma kp8-kept-valid (rewrite)
  (implies (and (ahn p q x)
                (memi q x)
                (kp8 q x) )
           (kp8 q (step p nd x)) )
  ((disable kp8 memi ahn)
   (use (kp8-eq) (kp8-dif)) ) ) ; [0.0 0.5 0.0]
```

Fig. 19. The invariant kp8

with the constant (top) instead of the variable top, however, it could not have been proved by induction.

Another typical invariant is kp8 of Fig. 19 which expresses that $\mathrm{nx}.(h.Q) = i.Q$ after the assignment to $i.Q$ in command 23. Here, $h.Q$ and $i.Q$ stand for the private variables h and i of process Q. The invariance of kp8 is proved by means of a case distinction. Lemma kp8-eq shows that kp8 of process P is invariant under the actions of P. In lemma kp8-dif, it is shown that kp8 of Q is invariant when $P \neq Q$ performs a command. Predicate memi expresses that $20 \leqq \mathrm{pc}.Q < 31$ implies $mem.(i.Q)$. Predicate ahn expresses

$$9 \leqq pc.P < 12 \wedge 17 \leqq pc.Q < 31$$
$$\Rightarrow \mathrm{aa}.P \neq h.Q \wedge \mathrm{aa}.P \neq \mathrm{nx}.(h.Q)$$

This means that process P does not place an invocation at an address where process Q is working.

The final lemma of Fig. 19 combines the results of the previous lemmas. Alternatively, it is possible to submit this final lemma directly, without the preparation and the hints for "use" and "disable". When we did this, NQTHM made a huge case distinction and reported the time triple [0.0 166.1 246.2]. This shows that the case distinction yields a speed-up by a factor 17.

The two invariants kg0 and kp8 are relatively innocent. A more critical example is kp6 which expresses that address $i.P$ is waiting when process P effectively executes command 22. It is given by

$$pc.P = 22 \wedge \mathrm{nx}.(h.P) = 0 \Rightarrow \mathrm{wa}.(i.P)$$

The proof of invariance of kp6 needs the support of five other invariants.

In [16], we use the atomicity rule of [2] Theorem 6.26 to argue that it is allowed to regard commands 11 and 12 as one command. Such an argument cannot be used to convince the theorem prover. Since we wanted the prover to support the program as it is presented, we had to provide a different argument. The solution is to simulate $ss.P$ by a companion $vs.P$ with the value

$$vs.P = \textbf{if } pc.P = 12 \textbf{ then } ss.P \cup \{\mathrm{aa}.P\} \textbf{ else } ss.P \textbf{ fi}$$

This function can then be used in the invariants at those positions where the update action 12 would be late. Compare the use of vpl as described above in part 4.

Part 7. Invariants of value transfer and computation

In this part, we treat the values of the private variables y and z, and arrays inv, res and sta. A typical invariant is kq2 which asserts

$$pc.Q \in \{26, 27, 28, 29\}$$
$$\Rightarrow \langle \mathrm{sta}.(h.Q), \mathrm{inv}.(i.Q), y.Q, z.Q\rangle \in R$$

Part 8. Aggregation of invariants

In this part, we prove that the predicates dubbed as invariants in the preceding parts are indeed invariants. More precisely, we form a conjunction of (universal quantifications of) these predicates and prove that this conjunction is a strong invariant that implies its constituents. In this way, we obtain the strong invariant called globinvariant. It follows that the above mentioned predicates kkg0, kp0, nwah, kp8, ahn, memi, kp6 and kq2 are indeed invariants.

At first sight, this part may look trivial. It is an important verification, however, of the completeness of the system of constituent invariants.

Part 9. Proof obligations of correctness

This part serves as a first main summary. Using all preparations in the previous parts, we here fulfill our proof obligations for correctness, i.e., prove the lemmas given in Figs. 11, 12, 14, and also the lemmas for the safety of inv and res.

14 The proof of wait-free progress

Part 10. Predicates for bounded delay

The proof of bounded delay for process Q begins with the observation that, whenever process Q traverses from command 13, via 16, 17 or 30, back to 13 again, variable gg is modified at least once (but not necessarily by process Q). For this purpose, we define predicate gstep to mean that some process, say P, executes command 30 with gg $= h.P$.

We construct a state function nrgsteps, initially 0, that is incremented whenever some process performs a gstep. This function is a companion of $\#\sigma$, the length of σ. It is defined by

$$\mathrm{nrgsteps} = (\mathrm{gphase} \oplus (\#\sigma)) - 1$$

where gphase is the function defined in Fig. 16 and operator $\oplus$ is conditional incrementation given by

$$b \oplus x = \textbf{if } b \textbf{ then } x + 1 \textbf{ else } x \textbf{ fi}$$

We then prove that every gstep has the effect that the state function seq.gg is incremented by 1 modulo n. It follows that, invariantly, seq.gg equals nrgsteps modulo n. Since seq.gg is the default process to be given priority for invocation treatment, this is the key step in the proof that individual starvation does not occur.

Part 11. Bound functions and bounded delay.

The next value for nrgsteps at which process Q is to be scheduled by default is defined as wupb.Q; it is the least number y such that

$$Q = y \textbf{ mod } n \wedge$$
$$(\mathrm{nrgsteps} < y \vee (\mathrm{nrgsteps} = y \wedge \mathrm{kw01}))$$

Here kw01 expresses that the default process is still schedulable. More precisely, it is the condition

$$(\mathrm{nx.gg} = 0 \vee \mathrm{nx.gg} = \mathrm{aa}.(\mathrm{seq.gg}))$$
$$\wedge (\forall T \in process :: h.T = \mathrm{gg} \wedge 20 \leqq pc.T < 31$$
$$\Rightarrow i.T = \mathrm{aa}.(\mathrm{seq.gg}) \wedge pc.T \neq 21)$$

Remark. In our mental proof, cf. [16], we had not seen the necessity of the conjunct $pc.T \neq 21$ at the end of kw01. This was one of the rare instances that the prover signaled a human error. If it had not been for this error, we would not have mentioned predicate kw01. (End of remark)

The difference between nrgsteps and wupb.Q is a descending measure for the waiting time of process Q. More precisely, we construct vfg.Q as the companion of this difference given by

$$(pc.Q > 14 \wedge \mathrm{gg} \neq h.Q) \oplus (\mathrm{wupb}.Q - \mathrm{nrgsteps})$$

We now prove that, while wa.(aa.Q) holds, i.e., while process Q is waiting, vfg.Q does not increase, and it decreases whenever Q makes a backward jump at 16, 17 or 30. We then define vfgg.Q as the number

if $pc.Q < 13$ then $n + 2$
elsif wa.(aa.Q) then vfg.$Q + 1$
elsif $14 \leqq pc.Q < 31$ then 1 else 0 fi

We define vlength.Q as

if $pc.Q < 4$ then $n - 1$ else #plist.Q fi

and vfloop.Q as the number

$$4 \times \mathrm{vlength}.Q + 18 \times \mathrm{vfgg}.Q + (32 - pc.Q)$$

It is proved that vfloop.Q is bounded by the constant (c22) $= 22 \times n + 65$, that it decreases at every command of Q apart from command 31, and that it descends at every command of another process. In this way, we obtain the results announced in Fig. 13.

15 Conclusions

The first conclusion is the correctness of the program of Fig. 4 with respect to the five proof obligations mentioned in Sect. 5. The input to NQTHM is a file of roughly 6500 lines. NQTHM needs around 75 minutes for the verification on a HP 9000-720.

The use of the prover enabled us to reach a finer grain of atomicity. For example, the sequential composition of commands 24, 25, 26 and 27 of the program is represented by one command in the program of [16]. Since four different shared variables are involved, this representation is not justified. The use of a mechanical prover has the advantage that it encourages a stricter separation between proof obligation and actual proof. This is especially useful when there are many proof obligations and long proofs. In [16], we did not yet have a good proof obligation for the safety of the non-atomic shared variables which occur in the commands 24, 25, 26 and 27. The proofs of safety, however, were straightforward extensions of the arguments of [16].

In the proof of progress, we followed the arguments of [16]. It soon became apparent, however, that our mental arguments rely heavily on our intuitive understanding of progress. In the end, it turned out that we obtained the proof obligation for progress as a side effect of the construction of the proof.

The initial predicate does not specify the initial values of array bb. This implies that the array inspection of nx.i in command 6 may be out of range. This point is not found by the mechanical proof since the prover works with an untyped language and all its functions are total. Actually, if bb.Q has not been set by process Q, the values $i =$ bb.Q and nx.i are superfluous in $ss.Q$. It can be proved that, if the initial predicate is strengthened with $mem.$(bb.T) for all processes T, then all array inspections and modifications are within range and the arrays wa, nx, sta, res, inv and seq are not inspected and modified at address 0.

One may ask whether the use of a mechanical theorem prover was justified: would not a careful reworking of the paper proof have yielded the same increase in confidence and insight as the effort to use the prover? In our view, the prover was indispensable to handle the amount of case distinctions needed for this method of proof. A satisfactory handwritten proof would require totally new insights to eliminate most of the case distinctions. Indeed, we would prefer such a result, but we do not see how to get it.

Let us finally answer the question whether we were satisfied with the capabilities of the NQTHM theorem prover. This question gains perspective by the wishes and requirements mentioned in [19] and [24]. We had some problems with the arithmetical weakness of NQTHM. This may be due to our isolated activity: we did not use or consult the existing libraries. NQTHM's lack of higher order functions did not matter much: for the problem at hand, low level reasoning was necessary anyhow, because of the potential interleavings. After some time, we found the LISP-like syntax and the lack of infix operators comfortable to live with. In conclusion, we think that, for the problem at hand, NQTHM was a good prover. It seems likely that, if we would have to start all over again, a new proof would be obtained much more quickly.

Acknowledgements. I would like to thank Jan Eppo Jonker and Rutger M. Dijkstra for their stimulating questions and criticisms about various versions of this paper, J. Moore for his ability and willingness to explain and solve the problems I had with NQTHM, and two anonymous referees for their suggestions that have led to a much better presentation.

References

1. Anderson JH, Grošelj B: Beyond atomic registers: bounded wait-free implementations of nontrivial objects. Sci Comput Program 19: 197–237 (1992)
2. Apt KR, Olderog E-R: Verification of sequential and concurrent programs. Springer, Berlin Heidelberg New York 1991
3. Ashcroft EA, Manna Z: Formalization of properties of parallel programs. Machine Intelligence 6. University of Edinburgh Press, Edinburgh 1971, pp 17–41
4. Back RJR, Sere K: Stepwise refinement of action systems. In: van de Snepscheut JLA (ed) Mathematics of program construction. Lect Notes Comput Sci, vol. 375 Springer, Berlin Heidelberg New York 1989, pp 115–138
5. Baeten JCM (ed): Applications of process algebra. Cambridge University Press 1990
6. Bevier WR, Hunt WA, Moore JS, Young WD: An approach to systems verification. J Autom Reasoning 5: 411–428 (1989)
7. Boyer RS, Moore JS: A computational logic. Academic Press, Orlando 1979
8. Boyer RS, Moore JS: A computational logic handbook. Academic Press, Boston 1988.
9. Chandy KM, Misra J: Parallel program design, a foundation. Addison-Wesley 1988
10. Dijkstra EW: Co-operating sequential processes. In: Genuya F (ed): Programming languages (NATO Advanced Study Institute). Academic Press, London New York 1968, pp 43–112
11. Gordon MJC: HOL: A proof generating system for higher-order logic: In: Birtwistle G, Subrahmanyam PA (eds) VLSI specification, verification and synthesis. Kluwer Academic Publishers 1988
12. Herlihy MP: Impossibility and universality results for wait-free synchronization. In: Proc 7th Annual ACM Symposium on Principles of Distributed Computing, August 1988
13. Herlihy MP: A methodology for implementing highly concurrent data structures. In: 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. SIGPLAN Notices 25(3): 197–206 (1990)
14. Herlihy MP: Wait-free synchronization. ACM Trans Program Lang Syst 13: 124–149 (1991)
15. Herlihy MP, Wing J: Linearizability: a correctness condition for concurrent objects. ACM Trans Program Lang Syst 12: 463–492 (1990)
16. Hesselink WH: Wait-free linearization with an assertional proof. Distrib Comput 8: 65–80 (1994)
17. Hunt WA Jr: FM8501: A verified microprocessor. Ph.D. Thesis, University of Texas at Austin, 1985
18. Hoare CAR: Communicating sequential processes. Prentice Hall 1985
19. Lindsay PA: A survey of mechanical support for formal reasoning. Softw Eng J, pp 3–27 (1988)
20. Milner R: A calculus of communicating systems. Lect Notes Comput Sci, vol 92. Springer, Berlin Heidelberg New York 1980

21. Misra J: Loosely-coupled processes. In: Aarts EHL, Van Leeuwen J, Rem M (eds): Parallel architectures and languages europe, vol 2. Lect Notes Comput Sci, vol 506. Springer, Berlin Heidelberg New York, pp 1–26
22. Moore J: A formal model of asynchronous communication and its use in mechanically verifying a biphase mark protocol. Formal Asp Comput 6: 60–91 (1994).
23. Owicki S, Gries D: An axiomatic proof technique for parallel programs. Acta Inf 6: 319–340 (1976)
24. Rushby JM, von Henke F: Formal verification of algorithms for critical systems. IEEE Trans Software Eng 19: 13–23 (1993)
25. Russinoff DM: A verification system for concurrent programs based on the Boyer-Moore prover. Formal Asp Comput 4: 597–611 (1992)
26. Plotkin SA: Sticky bits and universality of consensus. In: Proceedings of the 8th ACM Symposium on Principles of Distributed Computing 1989, pp 159–176