Proving correctness of two wait free shared data objects

Master Thesis

Department of Computer Science
University of Groningen

by Brandt Wijbenga

Supervised by W.H. Hesselink and H. Bekker

August 30, 2007

Proving correctness of two wait free shared data objects

Master Thesis by Brandt Wijbenga Department of Computer Science, University of Groningen Supervisors: W.H. Hesselink, H. Bekker – August 30, 2007

1 Preface

In this paper we present the research we have done for our master thesis. This research involved proving the correctness of two algorithms that construct a Load-Linked/Store Conditional data object from a Compare-And-Swap register and several atomic registers. We used an assertional method for the proof. This means that we have constructed proof goals and formulated predicates in assertional logic that express certain qualities of the algorithm that invariably hold (i.e. *invariants*). We also verified the proof using an automated theorem prover, PVS. Of the two proofs, only the first one has been completed, and we have discovered a minor algorithmic improvement. The proof for the second algorithm is incomplete – we were only able to verify its correctness by making a few assumptions.

The rest of this paper is ordered as follows. In the rest of this section, we will give an introduction that gives a slightly more detailed overview of our work. Since not all terms used in this introduction may be familiar, we follow this by some theoretical background, that hopefully covers enough ground so that the reader is not left in confusion. In section 2 we introduce how we approached the work and what notation we used. Next, in sections 3 and 4 we give the proofs of the algorithms. In section 5 we discuss how we used the theorem prover PVS in our work. Finally, a conclusion can be found in section 6 in which we look back at our work.

1.1 Introduction

In multi-threaded or multiprocessor systems, safe access to shared data is very important. Atomic read/write registers are not safe to rely on without locking mechanisms like mutexes or semaphores because of the danger of interference between concurrent processes that may lead to corruption of the data.

Compare-And-Swap (CAS) and load-linked/store-conditional (LL/SC) are instruction sets that are used to implement lock-free and wait-free atomic access to shared variables, without reading or writing corrupt data. The implementation of a concurrent object is said to be *lock-free* if some process is always guaranteed to make progress in a finite number of steps, and *wait-free* if *all* non-halted processes will make progress in a finite number of steps [7].

A CAS instruction atomically compares the current contents of a memory location with some relevant value and, if they match, modifies its contents. Often the 'relevant value' used for CAS is a copy of the memory location it acquired earlier.

A LL operation returns the current value of a shared variable, and a subsequent SC operation will only succeed in storing a new value if the shared variable has not been changed since the LL. If another process changed the shared variable in the meantime, the SC is guaranteed to fail. (As such, when compared to a CAS operation in which an older copy of the memory location is used, LL/SC is stronger than a CAS because of the guaranteed failure of a SC whenever the shared variable changes.)

P. Jayanti and S. Petrovic introduce several approaches to efficient implementations of LL/SC using 64-bit CAS or RLL/RSC (*restricted* LL/SC) objects and shared atomic registers in [11]. They argue that, even though LL/SC is the most

suitable set of instructions for the design of lock-free algorithms, most modern multiprocessors do not support those instructions but offer CAS or RLL/RSC instructions instead.

The method they use to building a LL/SC object using the CAS primitive is based on introducing sequence numbers to distinguish old data from new data. They present two algorithms, the first of which uses unbounded sequence numbers, and the second using a limited range of sequence numbers. They also give several correctness claims for both algorithms, which they prove in their full paper.

In my research I intend to show the correctness of the algorithms presented in [11] using a different proving method. Instead of behavioral proofs I constructed assertional proofs, a proving method based on defining invariants. While constructing the proof I used the proof assistant PVS to verify the proof, making sure the proof was correct. Proving correctness in this way is often a good approach to understanding the algorithm and it may also, if possible, reveal improvements that would not be obvious otherwise. For the first algorithm proved, such an improvement was indeed possible.

1.2 Theoretical Background

Concurrency Computers in the early days were (comparatively) primitive and could often do only one thing at the same time. However, as time and technology progressed, the technique of concurrency emerged. In concurrency, more than one actor (process or thread) simultaneously competes over shared resources. By 'applying' concurrency, functionality and performance of computer systems greatly increased, but it also gave rise to a whole new area of problems concerning concurrency control. When several things happen at the same time, it is important everything is happening correctly and in the right order without discrepancies. Concurrency problems were present in single processor systems: increasingly faster processors using interrupts and preemption allowed multiple processes or threads to run 'semi'-simultaneously, with interleaving. On the other side, in distributed systems several processes work concurrently with access to the same shared resources. More recently, multiprocessor systems have gained in popularity as well, with dualcore or quadcore (or more) computer systems. In all these systems concurrency is an issue.

Lock-based concurrency control To exercise some manner of control over concurrency, some kind of synchronization is needed. It is possible to do so by introducing locks. When using a lock-based approach, each section of a program that could raise problems when executed by more than one process concurrently is defined to be a critical section. Access to a critical section is controlled by a lock. A process can enter the critical section only when the lock is free, and it will claim the lock whenever it enters the critical section and release the lock upon exiting. Even though using locks can be useful at times, there are some major disadvantages to using locks. These disadvantages include blocking (processes have to wait for the lock to become free) and the possibility of deadlocks (two or more processes are waiting for each other to finish while holding a resource locked that the other process needs).

Lock-free concurrency control To avoid these problems there are alternatives that are lock-free. The CAS instruction, as said before, atomically compares the current contents of a memory location with some relevant value and swaps it for a new value. Since a CAS instruction takes place in a single atomic step, there are no problems with interference. However, an algorithm using a CAS operation can suffer from the so called *ABA-problem*. In short, the ABA-problem can be described as follows. Suppose that a process reads a shared value A. After this, a second process

changes the shared value to B and later back to A again. Even though the value has changed, when the first process now performs a CAS it will be able to do so successfully even though the value has changed in the meantime.

An alternative to CAS is LL/SC, specifically designed to avoid this problem. The LL/SC primitive consists of three procedures: a LL operation to read a value, a SC operation to store a value, and a VL operation to verify a current value. The LL operation is defined to return a valid value of the shared object, but this value is not necessarily the latest value. The SC operation is defined to succeed for a process if and only if the shared object has not changed in the time since the process last performed a LL operation.

Lock-free primitives in hardware Even though the LL/SC instruction set is more desirable than CAS, it is not as widely supported by hardware (or at least not yet) as the CAS instruction. The architectures that do offer LL/SC instructions mostly offer them in a weaker form because of architectural reasons, thus greatly reducing the advantages these stronger semantics have to offer. For this reason, several algorithms have been developed to construct a LL/SC object using the CAS primitive, some more efficient than others.

Program correctness There are different methods to prove that an algorithm is correct. One thing they all have in common is the large amount of work required. The field of formal verification has a long history. Originally, there was never much concern about formally proving correctness of programs. However, with concurrency emerging and the complexity concurrent programs brought, people began to pay attention to this issue, not only for concurrent programs but for sequential programs as well ([5],[6]). Hoare [9] introduced methods (like Hoare triples) to formally verify sequential programs (this method was extended for concurrent programs as well, see for example [12]). Dijkstra [4] introduced methods for program derivation that allowed development of algorithm and proof at the same time. Over the years techniques have been developed to improve upon the formal verification techniques, and especially [14] has been a major contribution to this field.

One approach to program correctness is to make some verification claims by doing model checking, but it is also a possibility to create a formal proof. We give a small overview here.

Model checking One can try to show correctness by creating a model from the algorithm and checking whether it satisfies a given formal correctness claim by going through all possible program states and executions methodically. Showing correctness by using a model checking tool is often impractical, especially for concurrent algorithms, because of a combinatorial increase of the state space (commonly known as the *state explosion problem*). Moreover, it is not always possible to create a finite model that can be exhaustively checked. Therefore it often happens that only a simplified model is checked, or only part of a problem. Still, model checking is widespread because it is easier to do than a formal proof, even though it will often not give a 100% correctness guarantee. It can also be a way to quickly reveal design errors.

Behavioural proof It is also possible to reason about algorithms behaviorally. Behavioral proofs typically involve arguments based on the order of different events. You try to show that at certain points in the algorithm, certain claims do (or do not) hold and thus your program has to be correct.

Assertional proof An assertional proof works differently: you try to construct a proof by forming formal predicates that make claims about the state of the algorithm at *any* given point. Because these predicates should invariably hold, they are also called *invariants*.

In general, assertional proofs are less common than behavioral proofs. Both methods have their advantages and disadvantages. Sometimes a behavioral statement is easier to come up with than a more abstract formal statement about a few variables. On the other hand, assertional proofs, because of their formality, tend to be complete and completely verifiable with the right tools. However, because of this attempted 'totality' they require more work to construct.

For proving concurrent algorithms, the (at least theoretically) preferred proving method seems to be an assertional proof: "behavioral proofs are unreliable and one should always use state-based reasoning for concurrent algorithms – that is, reasoning based on invariance.",[13]; "Most of behavioral proofs of concurrent programs are error-prone since it is difficult and tedious to take all possibilities of interleaving among the processes into consideration", [10]

After having said all that, the catch is that the two different methodologies actually aren't very different from each other. Hence, the choice between doing an assertional or a behavioral proof is often made subjectively, with key judgment factors probably being the familiarity or experience with the methodology. It is also not unusual to think up certain assertional statements after thinking about a problem behaviorally, or vice versa.

The proof method we used to verify the correctness of the algorithms is an assertional method, but in a slightly stripped down form.

2 Overview

Our goal is to show the correctness of two algorithms by Jayanti and Petrovic implementing a LL/SC object based on the CAS primitive. The first algorithm makes use of unbounded sequence numbers (and is therefore not unconditionally correct). The second algorithm restricts the sequence numbers and has no limitations. The proof of correctness of the first algorithm is complete, while the proof of the second is only partially complete but can be shown correct under certain assumptions.

2.1 Proof setup

The method used to prove the correctness of the algorithms is similar to the method in [8], and consists of 5 steps.

1. Provide an abstract specification of the LL/SC/VL algorithm. This specification describes the behaviour of the different functions.
2. Rewrite the original implementation, using rewriting rules to reach the smallest required grain of atomicity as (practically) possible. The rewritten algorithm should both implement the original algorithm as well as the formal specification.
3. Formulate proof obligations. What conditions do you have to satisfy in order to claim you have proved correctness?
4. Construct a proof for the proof obligations.
5. Verify the proof using the automated theorem prover PVS.

In steps 1 and 2 we will use *ghost variables* (also sometimes referred to as *auxiliary variables* or *history variables*)([1], [3], [14]). Ghost variables are variables that we add to the algorithm in order to help us prove correctness. They have no influence on the algorithm itself but are used to store additional information, allowing us to use them in specifications and in proofs at a more abstract level, showing the connection between the specification and implementation without influencing the program.

Steps 4 and 5 are intertwined. It will often be the case that when we suspect an invariant we will immediately try to prove it using the theorem prover. An advantage

of this method is that it can help speed up the process of determining whether the suspicion is true or false (especially in the 'false' situation) and if true, what other information may be required to complete a proof.

2.2 Notation notes

Throughout this paper, a standard convention is used to denote the different types of variables: shared variables will be written in `typewriter` font and private variables in a *slanted* font.

In [11], the notation of process identifiers is done by using subscript to indicate which process a variable belongs to. This notation was copied in figure 2. While this notation is acceptable, in our proof we will frequently encounter nested process identifiers. To maintain some level of readability without having to resort to sub-subscript notation we will use a different notation method. To indicate the owner of a private variable, we follow up the variable name by a period and a process identifier. For example, a process p's program counter pc indicating the position of the process in the algorithm will be written as $pc.p$. This way we can nest process identifiers like $pc.(q.p)$ to indicate the program counter of the process referred to by process p's variable q. We will sometimes omit the process identifier for local variables if there is no specific process to refer to, or when the associated process is clear already (i.e. in the code).

Single writer, multi reader shared variables will be treated as an array. For example, in the first algorithm, we will refer to process p's old sequence number by `oldseq`$[p]$.

On the naming of invariants The naming of invariants in the first (unbounded) algorithm is different from the second (bounded) algorithm. In the first proof we only number the predicates. In the second proof we also add letters. The reason for this discrepancy is that the bookkeeping in PVS (especially reordering and renaming of predicates) is easier to do when not working with just numbers – something we learned along the way. It would have been more aesthetic if we had changed the naming of invariants in the first algorithm afterwards, but this requires a lot of work that we rather spent on proving the second algorithm. As for the naming in the second proof, note that this is not consistent everywhere. This is due to the fact that we did not completely finish the second proof and did not reorder the predicates.

3 The unbounded algorithm

The first algorithm we describe and analyse is an implementation of LL/SC by means of CAS and unbounded sequence numbers. Technically, the term 'unbounded sequence number' is not correct because in practice there is no such thing as an unlimited data type to store this number. This is a known issue – the authors have intentionally built the first algorithm this way. They argue that in practice, this limitation not a large issue because of the way it is designed; it would be very unlikely to see problems as a result of this. We write a bit more on that in section 4.

3.1 Description of the unbounded algorithm

We have presented the original unbounded algorithm from Jayanti and Petrovic's design in figure 2. We first give a short description of how it works. A more detailed informal description can be found in [11].

The algorithm is set up for N processes. The shared variable X is central to the algorithm. It does not store the actual values that processes try to write, but rather provides a pointer [X.*pid*, X.*seq*] (a process number and a sequence number) that indicates the location of the last stored data value. Each process p has four shared atomic[1] single-writer, multi-reader registers $\texttt{val}_p[0]$, $\texttt{val}_p[1]$, $\texttt{oldval}_p$ and $\texttt{oldseq}_p$, and the local *persistent* variables tag_p and seq_p, whereby 'persistent' means that the variables retain their value between subsequent calls. The variable declarations can be seen in figure 1.

Types:

valuetype	64-bit number
seqnumtype	$(64 - \log N)$-bit number
processtype	number from range $0 \ldots N-1$
tagtype	**record** *pid*: processtype; *seqnum*: seqnumtype **end**

Shared:

X	tagtype
$\texttt{val}_p[0], \texttt{val}_p[1], \texttt{oldval}_p$	valuetype
$\texttt{oldseq}_p$	seqnumtype

Private:

tag_p	tagtype
seq_p	seqnumtype

Fig. 1. Data types and variable declarations of the unbounded algorithm

Each process p has a sequence counter seq_p that is incremented with each successful SC it performs. It is used to indicate the number of p's next SC. The current value of the abstract variable is preserved in $\texttt{val}_{\texttt{X}.pid}[\texttt{X}.seqnum \bmod 2]$. The register $\texttt{oldval}_p$ contains the previous value stored by p, and $\texttt{oldseq}_p$ contains the previous sequence number. Process p stores in tag_p a local copy of X that it acquires at the moment it starts a LL.

Note that the definition of 'seqnumtype' makes the (supposedly unbounded) sequence number actually a $(64-\log N)$-bit number. Strictly speaking, the algorithm would be incorrect because of this. In the proof of the algorithm however, we will replace this with the (abstract) data type 'int', that we assume to be unlimited, to avoid this problem.

The algorithm, printed in figure 2, consists of three procedures, viz. LL, SC and VL. We discuss them in that order.

The LL operation starts by copying X to tag_p (note that tag_p is referred to as $[q, k]$). It then continues to read the value from the val register. Because reading X and val cannot be done in one atomic step, p has to make sure that the value it just read is still up to date. It does this by comparing k, supposedly q's current sequence number, to k' (that just got the value $\texttt{oldseq}_q$ in line 3). If k' is still smaller than k, the original value is returned because it's still safe to use. In the other case, the value read from val in line 2 may be outdated or even invalid, and the LL procedure will return $\texttt{oldval}_q$ which is guaranteed to be an outdated, but valid value.

Process p's SC operation first stores its argument in a val register. If the SC is going to succeed, X will point to this register after the completion of the SC operation. It then performs a CAS-operation on X using the tag_p it set during its LL. A CAS is well defined, and behaves according to the specification also given

[1] The authors are not conclusive in their paper here. They specify these registers both by *atomic* as well as *safe*. See section 3.6 for an elaboration on this.

```
     proc LL(p)
1:   tag_p = X
     [q, k] = [tag_p.pid, tag_p.seqnum]
2:   v = val_q[k mod 2]
3:   k' = oldseq_q
4:   if (k' = k − 2) ∨ (k' = k − 1) return v
5:   v' = oldval_q
6:   return v'

     proc VL(p)
14:  return X = tag_p

     proc SC(p, v)
7:   val_p[seq_p mod 2] = v
8:   if CAS(X, tag_p, [p, seq_p])
9:      oldval_p = val_p[(seq_p − 1) mod 2]
10:     oldseq_p = seq_p − 1
11:     seq_p = seq_p + 1
12:     return true
13:  else return false

     proc CAS(X, u, v) returns boolean
        if X = u then X := v; return true; else return false; end
```

Fig. 2. The unbounded algorithm and the behavior of a CAS operation

in figure 2. If the CAS fails, this means that another process has written X in the meantime and the SC fails as well. A successful CAS means that the *tag* was still valid and p's CAS has written a new X with $[p, seq_p]$. Process p continues by updating its `oldval` and `oldseq` registers, incrementing its sequence counter and exiting successfully.

The VL operation compares X to process p's current tag and returns *true* if the two still match.

Note As can be seen in the code, there are two locations from where a LL(p) can possibly return a value, in line 4 and in line 6. The return in line 6 is an old value and p is guaranteed to fail its subsequent SC operation. Therefore we will call this return an *unsuccessful* return, while a *successful* return will mean a return using a correct (at that moment) value in line 4.

3.2 Formal Specification

The formal LL/SC specifications we use are the same as the ones used in [8], called LLe/SCe/VLe. These specifications can be seen in figure 3.

The specification introduces a shared ghost variable `hist` that contains all SC-stored values, and a shared ghost variable `top` pointing to the last stored value, such that `hist(top)` is the current value referred to by X. This means that each successful SC increments `top` once.

The specification also introduces the private ghost variables *ll* and *start* that are used for recording the value of `top` at the moment of performing an LL operation. They are part of making sure that a process exiting a LL will return a value that is not too old.

In figure 3 we have added an extra column next to the list of specification variables in which we list the original variables they correspond with.

There is a difference important to note concerning the return values. We assume there is a global state of some kind, an environment in which the LL, SC and VL procedures take place. There where return values were used previously we now introduce the private variables v and *result* to store the results in. Furthermore, just like we promised earlier, we have eliminated the data type 'seqnumtype' and replaced it with 'int', assumed to be unlimited.

3.3 Rewriting the algorithm

To match our specification we have rewritten the original algorithm from figure 2. The resulting reformulated algorithm can be seen in figure 5.

In order to preserve atomicity, we have to maintain at most one read from or write to a shared variable in each step. They can be combined with as many operations on private variables and ghost variables (whether shared or private) as required, since these pose no threat of interference with the actions on shared variables.

```
Types:
   valuetype        64-bit number
Ghost vars:
   start, ll, top   int
   hist             array[int] of valuetype
Private vars:                    Corresponds with:
   v                valuetype     v, v' (from LL)
   arg              valuetype     v (from SC)
   result           boolean       SC's and VL's return value

proc LLe(p)
   start := top
   choose ll with start ≤ ll ≤ top ; v := hist(ll)

proc SCe(p)
   if ll = top then
      top++ ; hist(top) := arg ; result := true
   else result := false end

proc VLe(p)
   result := (ll = top)
```

Fig. 3. Formal specifications of LL, SC, and VL.

In addition to the new variables coming from the specification, we introduce two more ghost variables in figure 4. First, for each process p there is a single writer, multi reader register $\texttt{loc}[p, x]$. It contains old top pointers to the hist-register in order to keep track of which process stored which values (although the hist-register stores all old values, it does not record *who* stored them). Second, for each process p there is a 1-bit register $\texttt{ext}[p]$ that is used by processes other than p, indirectly to check whether p is currently at line 10 of the algorithm or not. That information is, as we will show later, required for retrieving the correct value of *ll*.

```
Ghost vars:
   loc      array[processtype, int] of int
   ext      array[processtype] of bit
```

Fig. 4. New variable definitions in the rewritten algorithm.

The LL/SC/VL procedures act the same as their original counterparts but we have added the ghost variables from the specification and rewritten the private variables that used a return value. Furthermore, we added a continuously looping function **calling**(p) that arbitrarily chooses between performing a LL, SC or a VL operation. This function can be regarded as the environment that uses the implemented LL/SC variable in unknown ways.

Discussion In line 1, we add assignments to the ghost variables *start* and *ll* as required by the specification. Lines 3 and 4 can be combined into line 3, eliminating the need for k', and the 'return'- statement has been replaced by a 'goto'-statement (recall that we removed the need for a explicit return by designating v as the return value). In line 5 and 6 this means v' is replaced by v and we add another 'goto'-statement in place of the original line 6's 'return'. We also add another assignment to *ll* in line 5, the value depending on ext.

```
        calling(p)
50:       ( goto 1  ‖  goto 14  ‖  choose arg , goto 7 )

        proc LL(p)
1:        start := top; tag := X; q := tag.pid; k := tag.seqnum; ll := loc[q, k];
2:        v := val[q, k mod 2];
3:        if ((oldseq[q] = k − 2) or (oldseq[q] = k − 1)) then goto 6; end if;
5:        v := oldval[q]; ll := loc[q, oldseq[q] + ext[q] ];
6:        goto 50;

        proc SC(p)
7:        val[p, seq mod 2] := arg;
8:        if (X = tag) then
            X := [p, seq]; top++; hist(top) := arg;
            loc[p, seq] := top; seq++; result := true;
          else result := false; goto 50; end if;
9:        oldval[p] := val[p, seq mod 2]; ext[p] := 1;
10:       oldseq[p] := seq − 2; ext[p] := 0; goto 50;

        proc VL(p)
14:       result := (X = tag); goto 50;
```

Fig. 5. The reformulated algorithm.

In line 8, the CAS operation is replaced by the CAS specification from figure 2, but without the explicit return. Instead of u and v we use tag and $[p, seq]$, respectively. The result of the CAS is made available in *result*, which is in turn also the result of the SC. Also in line 8, we add assignments to `top`, `hist(top)` and $\texttt{loc}[p, seq.p]$. Finally, p's seq counter increment has migrated here from line 11. This is allowed because it concerns an assignment to a private variable, but we then have to rewrite the assignments with $seq.p - 1$ in lines 9 and 10 to $seq.p - 2$. The **mod**-operation on seq in line 9 becomes $seq.p - 2$ **mod** 2 and can thus be eliminated. The result of the SC is made available in *result*. Lines 9 and 10 also include assignments to the ghost variable `ext` now.

It should be clear that the rewritten algorithm is in essence still the same as the original algorithm: the added ghost variables are only used to store extra information and have no influence on the algorithm or on the original variables. In addition, nothing of the moved or shortened code effects the original algorithm in any (malicious) way (more background on this "auxiliary variable transformation" can be found in [14], (3.7)).

Variable descriptions and initialization For the initialization step, we pretend that process 0 performed a successful "initializing SC" with an initial value $\mathcal{V}$. Then, for all processes p, numbers x, and bits b the initialization is performed as seen in figure 6.

Some of these initializations are "arbitrary": for the algorithm itself it does not matter which values are used initially, but for the proof it does matter because all invariance claims that we make must also hold in the beginning.

The ghost variables are used in the following ways. The `hist`-register stores all old values written with SC, and `top` is the pointer to the last stored value. Therefore the initialization sets them to $\mathcal{V}$ and 1 respectively, such that `hist(top)`

Ghost variables:	**Shared variables:**	**Private variables:**
$\texttt{hist}(1) = \mathcal{V}$;	$\texttt{X} = [0,1]$;	$tag.p = (0,0)$;
$\texttt{top} = 1$;	$\texttt{val}[p,b] = \texttt{oldval}[p] = \mathcal{V}$;	$seq.0 = 2$;
$\texttt{loc}[p,x] = x$;	$\texttt{oldseq}[0] = 0$;	$\forall p : p \neq 0 : seq.p = 1$;
$start.p = ll.p = 0$;	$\forall p : p \neq 0 :$	$result.p = false$,
$\texttt{ext}[p] = 0$;	$\texttt{oldseq}[p] = -1$;	$v.p = arg.p = \mathcal{V}$

Fig. 6. The initialization values

is the first stored value. The register $\texttt{loc}[p,x]$ keeps track of who stored what. The initialization of $\texttt{loc}[p,x]$ to x fits with process 0's "initializing SC" setting $\texttt{loc}[0,1]$ to 1. The rest of the initial loc values is arbitrary. Next, *start* and *ll* are given the values 0. Finally, $\texttt{ext}[p]$ is used as a helper variable that indicates to other processes whether p is in line 10 or not with a 0 or 1 value. Since none of the processes start in line 10, all ext-values are initially 0.

The shared variables behave as in the original algorithm. Remember that X stores a pointer to the *actual* data value that is stored locally. In the initialization case, the process with *pid* 0 stored a value with sequence number 1. The val-registers store the actual data values that X can point to. They are all initialized with $\mathcal{V}$, even though it is only required for $\texttt{val}[0,1]$. All oldval are also initialized to $\mathcal{V}$, an arbitrary value as well since there are no 'real' old values yet. The same also goes for oldseq, that should hold the old sequence numbers that are coupled to the oldval values. Only process 0 has set it to 0, the rest are still "arbitrary" on -1.

Then there are the private variables. Here too, $tag.p$ is a local copy of X that p acquires at the moment it starts a LL operation. The *seq*-counter gives the number of a process' *next* SC, which is 2 in the case of process 0, and 1 for the other processes. The other private variables initializations are trivial: v and *result* are the return values for the LL procedure and the SC and VL procedures, and $arg.p$ is the value p tries to store in X with an SC.

3.4 Proof obligations

In order to prove the algorithm correct, we need to show that the rewritten algorithm from figure 5 implements the specifications of LL/SC/VL we presented in figure 3. Specifically, we have to show that the variables in the algorithm correspond in some way to the specification variables at the right locations.

We can see that the LL specification ends with choosing values for *ll* and v. Our algorithm will implement that specification if it ends accordingly, i.e. :

$$\text{(Ob1.)} \qquad pc.r = 6 \;\Rightarrow\; v.r = \texttt{hist}(ll.r) \;\wedge\; start.r \leq ll.r \leq \texttt{top}$$

In addition, the specifications of SC and VL determine their outcome by comparing *ll* to top. In the algorithm this outcome depends on the comparison of X to *tag*. Thus, we can link the two together in the following predicate to show the ties between specification and implementation:

$$\text{(Ob2.)} \qquad pc.r \in \{8,14\} \;\Rightarrow\; (\texttt{X} = tag.r \;\equiv\; ll.r = \texttt{top})$$

These are the proof obligations we have to satisfy, and they should hold for every execution and process. In the proof, described in the next section, the first proof obligation will be met by combining invariants (19.) and (20.). The second proof obligation is achieved by invariant (34.).

3.5 Proof of Correctness

The construction and description of the proof follows our understanding of the algorithm. Instead of aiming for the proofs of (Ob1.) and (Ob2.) directly, we look at the smaller and more rudimentary building blocks of the algorithm first.

We start with a proof for the correct value of X, and continue with proving that when a LL returns an old value, a subsequent SC is guaranteed to fail. Next we will provide some proofs about the contents of the different registers. Finally we will show how the algorithm satisfies the formal specification by proving the obligations, using invariants that happen to be even more strongly formulated than actually required.

The description of the proof follows a top-down approach most of the time. We will first give a desired (or suspected) predicate, followed by a description of how it is or can be true, or why we want it to be true. Some predicates have their invariance threatened under certain conditions. If that is the case we will also list these threats and provide information about how these threats can be eliminated.

In the proof of this algorithm we have chosen the names of the different invariant to be simple numbers. In the theorem prover PVS we appended these numbers to the generic name 'inv' (for invariant). This means that in the PVS file, invariant (01.) is named `inv01`, invariant (02.) `inv02`, and so on.

The value of X The first main important thing is to ensure that the `val` register referred to by X holds the last stored value, i.e.:

(01.) $\quad \texttt{val}[\texttt{X}.pid, \texttt{X}.seqnum \bmod 2] = \texttt{hist(top)}$

Initially this invariant holds, because of the aforementioned initializations[2]. After that, it can be threatened either by a change to X or by a new value in $\texttt{val}[\texttt{X}.pid, \texttt{X}.seqnum \bmod 2]$.

We can see that X is only changed in line 8, but in the same atomic step both auxiliary variables `hist` and `top` are updated to reflect the new situation. We do however have to ensure that when a process executes line 8 and writes a new X, the `val`-register it points to already holds the correct value:

(02.) $\quad pc.r = 8 \;\Rightarrow\; \texttt{val}[r, seq.r \bmod 2] = arg.r$

This invariant is obviously correct since `val` was assigned in line 7 and $\texttt{val}[r, seq.r \bmod 2]$ is not modified by processes other than r.

We then need to show that (01.) is not invalidated by a change in `val` in line 7. This can be done by using the following predicate:

(03.) $\quad \texttt{X}.pid \text{ at } 7 \;\Rightarrow\; seq.(\texttt{X}.pid) \bmod 2 \neq \texttt{X}.seq \bmod 2$

This says that whenever the process that last wrote X wants to write a new argument in its `val`-register, it is not allowed to change the one `val`-register that X currently points to (because this would invalidate the data). The validity of predicate (03.) follows from:

(04.) $\quad seq.(\texttt{X}.pid) = \texttt{X}.seq + 1$

Invariant (04.) claims that the sequence number of $\texttt{X}.pid$ is one higher than the sequence number that it stored during its last SC. It is easy to see that this is always true because *seq* is incremented in line 8, the same line where X is written.

[2] In the proof we have made sure that all variables are properly initialized, thus all predicates we have formulated will hold initially. We choose to say this only once, instead of repeating it with every predicate.

Ensured SC failure after reading an old value A process that exits the LL procedure with an old value of X must be guaranteed to fail its subsequently attempted SC operation.

A LL will be unsuccessful when it fails the guard in line 3 and enters line 5 to read $\texttt{oldval}[q.r]$. This means that when a process r gets there, the tag read by r is not valid anymore, or:

(05.) $\quad pc.r = 5 \quad \Rightarrow \quad tag.r \neq \texttt{X}$

Note that $tag.r$ equals to $[q.r, k.r]$, and X can also be read as $[\texttt{X}.pid, \texttt{X}.seq]$. If $tag.r \neq \texttt{X}$ holds, it essentially means that $q.r$ is not the latest process to write X.

To see whether a $[q.r, k.r]$-pair is still 'valid' or not, we need to compare $k.r$ to process $q.r$'s *current* sequence number. Remember that $seq.r$ is the sequence number of r's *next* SC. Therefore, r's *last* SC had sequence number $seq.r - 1$. A sequence number smaller than this indicates a SC older than r's last SC. We can formulate this as:

(06.) $\quad a < seq.r - 1 \quad \Rightarrow \quad [r, a] \neq \texttt{X}$

This follows from (04.). However, keep in mind that we are not interested in the sequence number of an arbitrary process r but in the sequence number of the process $q.r$, compared to k. This can be instantiated in (06.) and it would read $k.r < seq.(q.r) - 1 \quad \Rightarrow \quad [q.r, k.r] \neq \texttt{X}$.

Now, if we can show that $k.r < seq.(q.r) - 1$ holds in line 5, this means that we can use invariant (06.) to prove that (05.) holds. The obvious place to learn something about the values of $q.r$ and $k.r$ in line 5, is the if-statement in line 3 for process r (because r only enters line 5 when the guard of this if-statement fails). The guard in this if-statement compares $k.r$ to $\texttt{oldseq}[q.r]$, so we have to make some claims about these variables. It is easy to see that:

(07a.) $\quad pc.r \notin \{9, 10\} \quad \Rightarrow \quad \texttt{oldseq}[r] = seq.r - 2 \ \wedge$
$\quad\quad\quad pc.r \in \{9, 10\} \quad \Rightarrow \quad \texttt{oldseq}[r] = seq.r - 3$

(07.) $\quad \texttt{oldseq}[r] = seq.r - 2 \ \vee \ \texttt{oldseq}[r] = seq.r - 3$

(08.) $\quad k.r < seq.(q.r)$

Here, (07a.) is the result of the different locations in which $\texttt{oldseq}[r]$ and $seq.r$ are assigned, and (07.) is the simplified version of (07a.). Predicate (08.) is only threatened in line 1, but keeps valid as a result of (04.). We can take expressions (07.) and (08.) together to make the following claim:

(09.) $\quad k.r - 2 \leq \texttt{oldseq}[q.r]$

Knowing that this is true, this means that a failing guard in line 3 implies that invariant that (06.) holds. Therefore, (05.) holds too and $tag.r$ is indeed outdated. All of this can be illustrated by showing:

(10.) $\quad pc.r = 5 \quad \Rightarrow \quad k.r < seq.(q.r) - 1$

The validity of this follows from (07.) and (09.), since we want this to hold when r arrives at 5. By (09.) we know that $k.r - 2 \leq \texttt{oldseq}[q.r]$, but to get to line 5 the guard in line 3 should fail and thus $\texttt{oldseq}[q.r] \neq k.r - 2$ and $\texttt{oldseq}[q.r] \neq k.r - 1$. Therefore $k.r \leq \texttt{oldseq}[q.r]$, and applying (07.) to this inequality it can be read as $k.r \leq seq.(q.r) - 2$, which is $k.r < seq.(q.r) - 1$.

By taking (06.) and (10.) together, we can prove predicate (05.) by implication.

Contents of various registers For the first proof obligation (Ob1.), our goal is to show that the value of *ll* chosen in line 1 or 5, and the value of v chosen in line 2 or 5, satisfy our specification of the LL/SC operation. Because the *ll* value is connected with the return value v, we have to look at the `val` and `oldval` registers. The `loc` register plays an important role here by storing older values that may already be out of scope of the real algorithm. That is why it is useful to turn our attention to `loc` first.

The `loc`-register Looking at the behavior of the `loc`-registers, we can see that $\texttt{loc}[r]$ is an ordered list of increasing values: a new value is written to loc only at position $seq.r$, and the value written is always `top`. Both $seq.r$ and `top` are variables that can only ever increase. Intuitively this is easy to see, but we need to prove it too. This claim can be formalized as follows:

(11.) $\quad a < b < seq.r \;\Rightarrow\; \texttt{loc}[r,a] < \texttt{loc}[r,b]$

Since $\texttt{loc}[r]$ is undefined from position $seq.r$ onwards, we add the extra condition for a and b to be smaller than $seq.r$. To prove (11.), we need to show that a higher index is visited and written each time a process executes line 8. To ensure this, we use:

(12.) $\quad a < seq.r \;\Rightarrow\; \texttt{loc}[r,a] \leq \texttt{top}$

It is not difficult to see that (12.) is indeed true: if r writes `X`, $\texttt{loc}[r, seq.r]$ equals `top`, and immediately after that $seq.r$ is incremented. Invariant (12.) can now help to prove (11.) because it makes explicit that both a and b from (11.), smaller than seq, yield `loc`-values smaller than `top`.

Predicate (11.) can now be used to claim the following relations between `loc` and `top`:

(13.) $\quad \texttt{loc}[r, seq.r - 1] \leq \texttt{top}$
(13a.) $\quad \texttt{loc}[r, seq.r - 2] < \texttt{top}$
(13b.) $\quad \texttt{loc}[r, \texttt{oldseq}[r]] < \texttt{top}$

We know that (13.) is true: $seq.r$ is the number of r's next SC and $seq.r - 1$ the number of r's last SC, so $\texttt{loc}[r]$ in position $seq.r - 1$ contains a value equal to `top` or lower. Invariant (13a.) follows directly from (13.): $\texttt{seq}.r - 2$ is smaller than $seq.r - 1$ and therefore points to a `loc` value earlier in the list. By (11.) this value is smaller than `top`. The validity of (13b.) requires a little bit more work because we need the explicit expression for `oldseq` defined in (07.), but otherwise follows from (11.) and (13.) similarly.

The last claim about `loc` that we make here is the following:

(14.) $\quad \texttt{loc}[\texttt{X}] = \texttt{top}$

All variables in this invariant are only changed in line 8 which makes the proof for (14.) somewhat trivial.

The `val` and `oldval` registers It is now possible to make some claims about the content of `val` items using the previous invariants. Remember that (02.) already told us about the `val` position currently 'in use' for a next SC. We now formulate an invariant that can be used to refer to the value of the last SC:

(15.) $\quad \texttt{val}[r, seq.r - 1 \bmod 2] = \texttt{hist}(\texttt{loc}[r, seq.r - 1])$

It says that the sequence number (modulo 2) position in val of the last write holds the value it indeed *did* write according to our ghost variable **hist**. Predicate (15.) could become invalid by a write in **val**, which happens only in line 7. However, if r gets to line 7 it will write in $seq.r$ **mod** 2 and not in $(seq.r - 1)$ **mod** 2, which means there is no danger in that case. Predicate (15.) could also become invalid in line 8, but we can deduct that both the **hist** and the loc positions that are written are safe: (13.) tells us that $\texttt{loc}[r, seq.r - 1] \leq \texttt{top}$, and in line 8 top is first incremented before writing in **hist**. Therefore the position in **hist** that is written can not be the one in the invariant. The loc position is safe because it is written in position $seq.r$ and not in $seq.r - 1$. The final requirement for the proof of (15.) is that the value of the item that is written by a successful SC must be correct. By (02.) we see which val holds the argument and the changes (if any) in line 8 update the **hist** register to match this value.

There is another important thing to note about val. If a process writes X and continues to lines 9 and 10, it is clear that the value in $\texttt{val}[r, seq.r$ **mod** $2]$ still holds a correct old value. This value is still needed to update $\texttt{oldval}[r]$ to the 'new old value'. We postulate:

(16.) $\quad pc.r \in \{9, 10\} \quad \Rightarrow \quad \texttt{val}[r, seq.r \textbf{ mod } 2] = \texttt{hist}(\texttt{loc}[r, seq.r - 2])$

This predicate tells us that immediately after a successful SC, that particular val register still holds the value of the previous SC. Validity is only threatened in line 8, but we can show that any changes here are harmless. Suppose that (16.) holds. Then, when a process r executes line 8, we can make a distinction between whether the 'threatening process' is r itself or another process.

If the threat is a process other than r, (13a.) ensures that the loc register that is accessed does not overwrite a previously stored history variable. If the threat is r itself, we can use (13.) instead of (13a.) to ensure the same thing. However, that is not enough yet with r being the threatening process: if r executes line 8, $seq.r$ is incremented and we should check whether the value in val satisfies (16.). This is where we can use invariant (15.), that shows us the value of $\texttt{val}[r, seq.r - 1$ **mod** $2]$. With one increment of the sequence counter we know that $\texttt{val}[r, seq.r - 2$ **mod** $2] = \texttt{hist}(\texttt{loc}[r, seq.r - 2])$ holds. The -2 part is eliminated by the **mod** 2-operation, and we are left with the consequent of (16.).

Now we can concentrate on the value of oldval. Ideally we would want to claim that $\texttt{oldval}[r] = \texttt{hist}(\texttt{loc}[r, \texttt{oldseq}[r]])$. Unfortunately, that is not always true since oldval is updated in line 9 and oldseq in line 10. That's why we make the following distinction:

(17.) $\quad pc.r = 10 \quad \Rightarrow \quad \texttt{oldval}[r] = \texttt{hist}(\texttt{loc}[r, seq.r - 2])$
(18.) $\quad pc.r \neq 10 \quad \Rightarrow \quad \texttt{oldval}[r] = \texttt{hist}(\texttt{loc}[r, \texttt{oldseq}[r]])$

One can see that (17.) holds initially, because in line 9 we are told by (16.) that the correct value is assigned to $\texttt{oldval}[r]$. Validity is preserved in line 8 by (13a.), that ensures the older values in **hist** are never disturbed.

In the same way you can use (13b.) to show that the old values for (18.) are untouched in line 8, using (07.) to substitute the value oldseq; also, because (17.) was has been shown valid, we can use it for the validity of (18.) in line 10.

The first proof obligation To satisfy the LL specification we need to choose a ll with $start \leq ll \leq \texttt{top}$ and $v = \texttt{hist}(ll)$. Invariants (19.) and (20.) show the conditions that must hold when exiting the LL procedure, and are in fact stronger variants of the proof obligations in (Ob1.):

(19.) $pc.r = 6 \quad \Rightarrow \quad v.r = \texttt{hist}(ll.r)$
(20.) $start.r \leq ll.r \leq \texttt{top}$

We first concentrate on proving (20.), and after that move on for the proof of (19.).

Validity of (20.): the range of *ll* Technically, predicate (20.) consists of two inequalities. Instead of proving both inequalities at once, it is easier to split the proof into two parts, one for each bound. This leads to the following invariants:

(21.) $start.r \leq ll.r$
(22.) $ll.r \leq \texttt{top}$

The lower bound We can see that invariant (14.) ensures the validity of (21.) through line 1. To show its validity in line 5 is somewhat more complicated, and we start with finding another expression for $start.r$. Notice that $start.r$ is assigned a value in line 1, along with $q.r$ and $k.r$. This means that initially, at the time of assigning, $\texttt{loc}[q.r, k.r] = \texttt{top} = start.r$ (because of (14.)). Since we 'know' that $start.r$ and $\texttt{loc}[q.r, k.r]$ remain unchanged we claim:

(23.) $start.r = \texttt{loc}[q.r, k.r]$

Since a change in $\texttt{loc}[q.r, k.r]$ could make (23.) invalid we still have to prove that what we 'know' about it is correct. The only way that $\texttt{loc}[q.r, k.r]$ can be changed is an execution of line 8 by process $q.r$ when $seq.(q.r)$ is equivalent to $k.r$. However, this can never be the case: invariant (08.) provides us with the knowledge that $k.r < seq.(q.r)$, and this means that $\texttt{loc}[q.r, k.r]$ can never be overwritten.

The expression for $start.r$ in terms of loc is useful because we can now compare it to the possible values of *ll* assigned in line 5, values that are also expressed in terms of loc. This is just the information we need to check (21.)'s validity through line 5. We formulate the following statement that, if true, will ensure exactly this:

(24.) $pc.r = 5 \quad \Rightarrow \quad start.r \leq \texttt{loc}[q.r, \texttt{oldseq}[q.r]] \ \wedge \ start.r \leq \texttt{loc}[q.r, \texttt{oldseq}[q.r] + 1]$

First of all note that if $start.r \leq \texttt{loc}[q.r, \texttt{oldseq}[q.r]]$ holds then $start.r \leq \texttt{loc}[q.r, \texttt{oldseq}[q.r] + 1]$ also holds as a result of predicate (11.). Therefore we focus only on the first conjunct (with $\texttt{oldseq}[q.r]$) in describing our proof of (24.).

Using (23.) to rewrite $start.r$ it is possible to read first conjunct of (24.) as $\texttt{loc}[q.r, k.r] \leq \texttt{loc}[q.r, \texttt{oldseq}[q.r]]$. As a result of (11.), this is true when $k.r \leq \texttt{oldseq}[q.r]$ holds, and therefore we want the following invariant to be true:

(25.) $pc.r = 5 \quad \Rightarrow \quad k.r \leq \texttt{oldseq}[q.r]$

As a result of (09.), predicate (25.) is true when r enters line 5. The proof for this can be done in way similar to the proof of (10.) that was described in section 3.5. The preservation of (25.) is only threatened by a new assignment of $\texttt{oldseq}[q.r]$ in line 10, but from (07.) we know that the value $q.r$ writes in $\texttt{oldseq}[q.r]$ is safe to use.

We can now shift our focus up again and show that the invariance of (24.) is implied by predicates (25.), (07.), (11.) and (23.) in the following way:

$pc.r = 5$
$\quad \Rightarrow \quad \{(25.)\}$
$k.r \leq \texttt{oldseq}[q.r]$
$\quad \Rightarrow \quad \{(07.)\}$

$$
\begin{array}{l}
k.r \leq \texttt{oldseq}[q.r] < seq.(q.r) \\
\qquad \Rightarrow \quad \{(11.)\} \\
\texttt{loc}[q.r, k.r] \leq \texttt{loc}[q.r, \texttt{oldseq}[q.r]] \\
\qquad \Rightarrow \quad \{(23.)\} \\
start.r \leq \texttt{loc}[q.r, \texttt{oldseq}[q.r]]
\end{array}
$$

Since expression (24.) is hereby shown invariant, (21.) is indeed preserved through line 5.

The upper bound Proof of the validity of (22.) is easier than the proof of (21.). In line 1 we can use invariant (14.) to see that (22.) is preserved, much similar to the proof of (21.) through line 1. The validity of (22.) in line 5 is guaranteed too. First we point out that the value read into $ll.r$ is $\texttt{loc}[q.r, \texttt{oldseq}[q.r] + \texttt{ext}[q.r]]$. As a boolean value, $\texttt{ext}[q.r]$ is either 0 or 1, something that we formulate in (EXT.).

(EXT.) $\quad (\texttt{ext}[r] = 1 \quad \Rightarrow \quad pc.r = 10) \ \wedge \ (\texttt{ext}[r] = 0 \quad \Rightarrow \quad pc.r \neq 10)$

In the case where $\texttt{ext}[q.r] = 0$, we can use (13b.) to show the value is smaller than top. In the other case we can formulate (13c.) to show the same thing.

(13c.) $\quad \texttt{ext}[r] = 1 \quad \Rightarrow \quad \texttt{loc}[r, \texttt{oldseq}[r] + 1] < \texttt{top}$

This predicate is implied by (07a.), (13a.) and (EXT.).

With both (21.) and (22.) now proved correct, we have shown the validity of predicate (20.), one half of the first proof obligation.

Validity of (19.): values of v and ll We move on to the proof of invariant (19.). Recall the definition:

(19.) $\quad pc.r = 6 \quad \Rightarrow \quad v.r = \texttt{hist}(ll.r)$

If we want to say something meaningful about the values of v and ll, we first need an impression about the different possibilities. Looking at the algorithm, you can see that in case of a successful return, the variable $v.r$ is set to $\texttt{val}[q.r, k.r \bmod 2]$. This value is preserved in the ghost history variable at $\texttt{loc}[q.r, k.r]$. This is the same value assigned to $ll.r$ in line 1.

An unsuccessful return will read the $v.r$ value in $\texttt{oldval}[q.r]$ in line 5, and we can refer to it in either $\texttt{loc}[q.r, \texttt{oldseq}[q.r]]$ or in $\texttt{loc}[q.r, \texttt{oldseq}[q.r] + 1]$. The reason for this difference is the updating of oldval and oldseq in different lines. The $ll.r$ value assigned in line 5 corresponds with this value, making use of the other auxiliary variable ext. Note: this claim *only* holds at the time of r's execution of line 5, since $\texttt{oldseq}[q.r]$ may be changed later.

Apparently, the values for the variables here are different depending on whether the return is successful or not. That is why we split the proof for (19.) into two parts as well: one in which we cover the successful return, and one dealing with an unsuccessful return.

Successful return Before continuing, note that a successful return will evaluate the if-statement in line 3 to true. For reasons of readability we abbreviate the guard "$\texttt{oldseq}[q.r] = k.r - 2 \ \vee \ \texttt{oldseq}[q.r] = k.r - 1$" to "$guard_3.r$" in this section.

For a successful return, it is useful to formalize the following obvious invariant:

(26.) $\quad pc.r \in \{2, 3\} \quad \Rightarrow \quad ll.r = \texttt{loc}[q.r, k.r]$

Validity of (26.) is threatened when process $q.r$ executes line 8, but by (08.) $k.r < seq.(q.r)$, and we can see that the `loc` position that is written is not dangerous to (26.).

We can also see that passing the guard leads to the following invariants:

(27.) $\quad q.r \notin \{9, 10\} \ \wedge \ guard_3.r \ \Rightarrow \ k.r = seq.(q.r) - 1$

(28.) $\quad q.r \in \{9, 10\} \ \wedge \ guard_3.r \ \Rightarrow \ k.r = seq.(q.r) - 1 \ \vee \ k.r = seq.(q.r) - 2$

Formula (27.) follows from (07a.) and (08.), and (28.) follows from (07a.). The distinction by the location of $q.r$ is necessary because of the consequences this can have for the values in the registers, and will become more clear in the proof of (30.).

With (26.), we established the value of $ll.r$. For a successful LL, the $v.r$ return value used is the one assigned in line 2, namely $\texttt{val}[q.r, k.r \bmod 2]$, copied straight from $q.r$'s `val`-register. Then, if a process is in line 3 and about to return successfully because the guard holds, we can claim the following:

(29.) $\quad pc.r = 3 \ \wedge \ guard_3.r \ \Rightarrow \ v.r = \texttt{val}[q.r, k.r \bmod 2]$

To see the validity of (29.), consider this. If process r gets to line 3, it has just copied the values from the `val`-register and it should be true. However, it can be invalidated in line 7 and in line 10. If process $q.r$ executes line 7, invariant (27.) tells us the `val`-register that $q.r$ writes is 'safe' for r: it satisfies the consequent of the implication, so we know that $k.r = seq.(q.r) - 1$ and that means $\texttt{val}[q.r, seq.(q.r) \bmod 2]$ is still okay. If $q.r$ executes line 10, invariant (29.) is also still valid for r, this time as a result from (07a.) and (09.). This can be seen from the following reasoning. Assume that r is at line 3, and $q.r$ in line 10. There are two cases here to consider, one in which the guard is false and one in which the guard is true. If the guard is false, we have to show that it cannot become true when $q.r$ sets a new `oldseq`. By (09.) we know that $k.r - 2 \leq \texttt{oldseq}[q.r]$, and the guard was false so $k.r \leq \texttt{oldseq}[q.r]$. Substituting the expression with (07a.), this becomes $k.r \leq seq.(q.r) - 3$. Execution of line 10 by $q.r$ changes $\texttt{oldseq}[q.r]$ to $seq.(q.r) - 2$ and we can see that this does not make the guard true. Now consider the case where the guard is true. This leads once again to two other cases, one in which $\texttt{oldseq}[q.r] = k - 2$ and one in which $\texttt{oldseq}[q.r] = k.r - 1$. In the first case (07a.) lets us rewrite this as $k.r - 2 = seq.(q.r) - 3$, and after process $q.r$ executes line 10 the guard will stay true (since then $\texttt{oldseq}[q.r]$ becomes $k.r - 1$), without making changes to the consequent of the implication. The second case can be rewritten to $k.r - 1 = seq.(q.r) - 3$, and execution of line 10 by $q.r$ will make the guard invalid.

With an expression for both $ll.r$ and $v.r$, we are almost done. The only thing left is showing that the `loc`-register from (26.) corresponds with the `val`-register from (29.). Then we can tie those two invariants together and prove the 'successful' part of predicate (19.). Thus, we claim:

(30.) $\quad guard_3.r \ \Rightarrow \ \texttt{val}[q.r, k.r \bmod 2] = \texttt{hist}(\texttt{loc}[q.r, k.r])$

To prove this, we need two new invariants, (31.) and (32.), which are the results of $q.r$ applied to (15.) and (16.), respectively:

(31.) $\quad k.r = seq.(q.r) - 1 \ \Rightarrow \ \texttt{val}[q.r, k.r \bmod 2] = \texttt{hist}(\texttt{loc}[q.r, k.r])$

(32.) $\quad k.r = seq.(q.r) - 2 \ \wedge \ q.r \in \{9, 10\}$
$\qquad\qquad \Rightarrow \ \texttt{val}[q.r, k.r \bmod 2] = \texttt{hist}(\texttt{loc}[q.r, k.r])$

Combined with formulas (27.) and (28.), these formulas prove invariant (30.) by implication.

It is more apparent now why we choose to split the invariant for the value of $k.r$ into two different ones: we required information about the `val`-register deep enough to make claims about it as detailed as (32.)

Unsuccessful return For an unsuccessful return the value of $\texttt{oldval}[q.r]$ is used. We showed that the value of oldval depends on the position of the process in the algorithm. This information is already formulated in invariant EXT.

Using (EXT.) we can find out about the values of oldval assigned to $ll.r$. First, consider the case in which $\texttt{ext}[q.r]$ holds (i.e. equals 1). This means that $pc.(q.r) = 10$, and according to (17.), $\texttt{oldval}[q.r] = \texttt{hist}(\texttt{loc}[q.r, seq.(q.r) - 2])$. We also know that according to (07a.) $\texttt{oldseq}[q.r] = seq.(q.r) - 3$. Putting these two expressions together renders $\texttt{oldval}[q.r] = \texttt{hist}(\texttt{loc}[q.r, \texttt{oldseq}[q.r]+1])$. Similarly we can consider the case in which $\texttt{ext}[q.r]$ does not hold, i.e. equals zero. Then $pc.(q.r) \neq 10$, and according to (18.) $\texttt{oldval}[q.r] = \texttt{hist}(\texttt{loc}[q.r, \texttt{oldseq}[q.r]])$. In other words, combining the two cases yields exactly the expression: $\texttt{oldval}[q.r] = \texttt{hist}(\texttt{loc}[q.r, \texttt{oldseq}[q.r] + \texttt{ext}[q.r]])$. The loc-value herein is exactly the one assigned to ll in line 5.

Validity of (19.) Returning to (19.), we can ensure its invariance through line 3 using invariants (26.), (29.), and (30.): the guard holds, so by (30.), $\texttt{val}[q.r, k.r \bmod 2] = \texttt{hist}(\texttt{loc}[q.r, k.r])$, and invariants (26.) and (29.) give substitutions for the loc and val registers. The invariance through line 5 can be ensured by using (EXT.), (07.), (17.) and (18.) in the way we described it in the paragraph above. The only problem left is line 8, in which hist is changed. But we know that the changes are made in hist(top) and in (22.) we showed that $ll.r \leq \texttt{top}$, which makes the accessed hist safe to use. Thus, predicate (19.) is valid, and (19.) and (20.) together show the correctness of the first proof obligation (Ob1.).

The second proof obligation The remaining proof obligation (Ob2.) showing the link between program and specification variables can be justified as follows. According to the specification, a SC will succeed when $ll.r = \texttt{top}$, and a VL will return the result of $ll.r = \texttt{top}$. The algorithm uses the check $\texttt{X} = tag.r$. The relation between the specification and the algorithm is only complete if we can prove that there is an equivalence between the two expressions. This relationship is only required in lines 8 and 14, but we can show that it holds throughout the whole program. We formulate this in the following predicate:

(34.) $\quad ll.r = \texttt{top} \quad \equiv \quad \texttt{X} = tag.r$

Equivalence is proved by showing implications in both ways so, similar to (20.) we split this proof into two expressions:

(35.) $\quad \texttt{X} = tag.r \quad \Rightarrow \quad ll.r = \texttt{top}$

(36.) $\quad ll.r = \texttt{top} \quad \Rightarrow \quad \texttt{X} = tag.r$

Invariant (35.) is threatened in three places, in line 1 and 5, when giving $ll.r$ and $tag.r$ new values, and in line 8 where X and top can be modified. Line 1 is safe to execute: the $tag.r$ is set to X and $ll.r$ to $\texttt{loc}[q.r, k.r]$, which is then equal to $\texttt{loc}[\texttt{X}]$, and by (14.) also equal to top. Line 5 is safe because of (10.) and (06.): a process in line 5 is guaranteed to have $tag.r \neq \texttt{X}$. Finally, in line 8 invariant (08.) ensures that $k.r < seq.(q.r)$.

For invariant (36.) line 1 is not an issue. It is only threatened in lines 5 and 8. Using (13b.) we can guarantee that the value given to $ll.r$ in line 5 is smaller than top, and by (22.) $ll.r \leq \texttt{top}$ in line 8. It follows that execution of either line 5 or line 8 makes the antecedent of the implication invalid, which makes (36.) true.

Conclusion The proof of the first proof obligation (Ob1.) has been completed with showing the correctness of both predicate (19.) and predicate (20.). The second proof obligation (Ob2.) has been satisfied with the proof of predicate (34.). Having satisfied both proof obligations we can claim that the program is correct.

3.6 Observations

Safe vs. Atomic In [11] it is not entirely clear what kind of registers are used for the single writer, multi reader variables. They are referred to both as safe and as atomic. There is a small but important difference between the two. As might be expected from the name, *atomic* means that the registers can be read or written in one single atomic step. A little weaker, *safe* means that whenever a process reads the register while another process is writing it, the read returns an arbitrary (unknown) value of the right type. If it attempts to read the register while it is not being written, it correctly receives the last stored value.

In other words, if the registers are indeed atomic, there is no threat of interference or wrong values, but if they are "only" safe, we are faced with a few more proof obligations because of the problems this may give.

Remember, the registers in question are $\texttt{val}_p[0]$, $\texttt{val}_p[1]$, $\texttt{oldval}[p]$, and $\texttt{oldseq}[p]$. For those variables to be safe, the algorithm itself has to ensure that no interference is possible that may lead to incorrect behaviour. This leads to the following extra requirements:

(S1.) $pc.r = 2 \Rightarrow pc.(q.r) \neq 7 \vee k.r \bmod 2 \neq seq.r \bmod 2$
(S2.) $pc.r = 3 \Rightarrow pc.(q.r) \neq 10$
(S3.) $pc.r = 5 \Rightarrow pc.(q.r) \neq 9$

Of these three claims, the requirement (S1.) may seem stronger than needed: it is possible that even if the `val` value stored into v in line 2 is garbage, that it will be overwritten in line 5. However, this is not always the case since it depends on the result of the guard in line 3.

The predicates (S1.), (S2.) and (S3.) are not valid either: the LL/SC algorithm is designed to be lock-free and wait-free, and if process r is held up somehow in line 2, 3 or 5, it is perfectly okay for a process $q.r$ to keep on running and enter one of the lines that are prohibited by these implications. This suggests that all these registers must be atomic.

Improvement of the algorithm After having constructed the proof of correctness, we can see that a small improvement is possible in the algorithm.

In line 3 processes perform a check to determine whether the LL value they just read is still valid. The if-statement tests for $((\texttt{oldseq}[q] = k - 2)$**or** $(\texttt{oldseq}[q] = k - 1))$. But predicate (09.) tells us that $k.r - 2 \leq \texttt{oldseq}[q.r]$. This means that we can replace the test with $\texttt{oldseq}[q.r] < k$.

This has implications for the original algorithm as well: in figure 2 the LL procedure had to use an extra variable k' to store a copy of $\texttt{oldseq}[q.r]$ in (otherwise, $\texttt{oldseq}[q.r]$ might have to be read twice, with possibly resulting discrepancies). This variable is now obsolete, and a single, direct read of `oldseq` is possible. The resulting new (LL-)algorithm can be seen in figure 7.

```
        proc LL(p)
1:      tag.p := X;
        [q, k] := [(tag.p).pid, (tag.p).seqnum]
2:      v := val.q[k mod 2]
3:      if oldseq.q < k return v
4:      v' := oldval.q
5:      return v'
```

Fig. 7. The improved LL procedure

4 The bounded algorithm

As briefly stated before, the unbounded version numbers from the first algorithm eventually lead to a (theoretical) problem. The unbounded version numbers are stored in 'limited' registers. This causes the sequence numbers to wrap around eventually. Therefore the algorithm is not unconditionally correct. Even though this is not of real practical concern (the authors calculate that even with one million SC operations per second it would take 32 years for the sequence number to wrap around, making it very unlikely that an old value is incorrectly used), they also present an algorithm using bounded version numbers with a constant time complexity and a constant space overhead per process.

Their limited sequence number-implementation of the (64-bit) LL/SC object using (64-bit) CAS objects and (64-bit) registers consists of three steps. First, implementing a LL/SC object from a WLL/SC object; second, implementing a WLL/SC object from a 1-bit "pid" LL/SC object; and last, implementing a 1-bit "pid" object from a CAS object and registers.

A 1-bit "pid" LL/SC object is just like a normal 1-bit LL/SC object but its LL operation (conveniently called `BitPid_LL`) does not only return the 1-bit value but also the process identifier of the process that stored this returned value. For more details on the first two reductions we refer to the descriptions in [11]. Our focus is on the third and last reduction, the construction of the `BitPid_LL` procedure out of a CAS object and registers.

It might seem strange to be handling with 1-bit values now. The reason for this is the second reduction. Similar to the unbounded LL/SC algorithm, it stores the actual data values in a local (single writer, multi reader) register, and uses a 1-bit value to distinguish between them. When performing an SC, it will only store its process identifier and the value of this bit in the shared variable, knowing that other processes can look up the actual data value using this information.

In this section we give the partial proof of this bounded algorithm and the assuptions that we made to complete the proof.

4.1 Description of the bounded algorithm

We first look at the original version of the bounded algorithm, presented in figure 9. Its variables and type declarations are shown in figure 8.

Types:

valuetype	bit
seqnumtype	$(63 - \log N)$-bit number
processtype	number from range $0 \ldots N-1$
tagtype	**record** *seq*: seqnumtype; *pid*: processtype; *val*: valuetype **end**

Shared:

`X`	tagtype
`A`	**array** $[0 \ldots N-1]$ **of** tagtype;

Private:

old_p, chk_p	tagtype
v	valuetype
$seq_p, v_p, nextStart_p$	seqnumtype
$procNum_p$	processtype

Fig. 8. Data types and variable declarations of the bounded algorithm

In the type declarations, the valuetype has changed to single bit number (the `BitPid_LL` handles 1-bit values). The processtype is still a number below N, leaving room for the sequence numbers stored in a seqnumtype of $63 - \log N$ bits. The tagtype has changed to a record with three values: a sequence number, a process identifier, and a data value.

The bounded algorithm works with a shared variable `X` again, but also introduces a shared array `A` that is used for announcing LL operations. This announcement array is used as follows. Processes that perform a LL operation will, after reading `X`, write the (process identifier, sequence number)-pair from the `X` they read to the announcement array. While this entry is still in the announcement array, it will prevent the process associated with the process identifier from reusing that particular sequence number. It is not *safe* to reuse while it is still in the announcement array. This will become clearer when we discuss the procedures in detail.

Each process has a range of new private variables. First, *old* and *chk* are used to store local copies of `X`. Next, v is used for the argument of the SC procedure. The sequence number is stored in *seq*, and the variables *val*, *nextStart*, and *procNum* are used to select new sequence numbers.

For a process p, the LL procedure first reads `X` and stores it in the local variable *old*. It then uses the array `A` to announce that it has read `X` by storing $old_p.seq$ and $old_p.pid$ in `A`$[p]$. The third argument (the 0) stored in line 2 is not relevant for now as it has something to do with selecting new sequence numbers. Process p continues in line 3 by making another copy of `X` in *chk*, and then returns the values read in *old* in line 4.

Process p's SC procedure first checks whether *old* equals *chk* or not. If they differ, this means the value in `X` has been changed even during p's LL operation, and the SC must fail. In line 6 it attempts to store the argument v with a CAS operation, along with p's pid and sequence number that, taken together, act like a tag. If the CAS fails, `X` has also changed and the SC returns reporting failure. If the CAS succeeds, a new `X` has been put in place and p continues in line 8, choosing a new sequence number using the function `select`, and returns in line 9.

At first glance, the second read of `X` in line 3 seems superfluous. This is not the case since the read of `X` in line 1 and the announcement in `A` thereof in line 2 both involve an operation on a shared variable. Therefore they can not be combined into one single atomic step. The second read will prevent errors in satisfying the LL specification. If a (process p's) second read of `X` gives the same value as the first read, then this value has already been announced in line 2 and p's read of `X` appears to have been atomic. If the second read of `X` gives a different value than the first read, one or more SC operations have taken place while p was executing its LL; an old value is returned and p can ensure that its subsequent SC operation will fail by comparing its *old* and *chk* values.

The *read* procedure included in figure 9 is a result of the abstraction. It is an atomic read instruction for `X`. Because of the simplicity of the *read* procedure we will not take it into account in our proof.

A final of note is that the `select` function requires a constant amount of space per process and runs in constant running time.

Selection of a new sequence number The authors present two selection algorithms, a "simple selection" algorithm and a "efficient selection" algorithm. In this paper we only treat the simple selection algorithm. This is the reason why the third argument in line 2 of the algorithm is irrelevant, as it is a part of the efficient selection function, not part of the simple selection function.

```
    proc BitPit_LL(p)                    proc SC(p, v)
1:  old_p = X                        5:  if (old_p ≠ chk_p) return false
2:  A[p] = [old_p.seq, old_p.pid, 0] 6:  if ¬CAS(X, old_p, [seq_p, p, v]) return false
3:  chk_p = X                        7:  seq_p = select(p)
4:  return [old_p.pid, old_p.val]    8:  return true

    proc VL(p)                           proc read(p)
9:  return (old_p = chk_p = X)       10: tmp = X
                                     11: return [tmp.pid, tmp.val]
```

Fig. 9. The bounded algorithm

We have presented the simple selection algorithm in figure 10. The basic principle behind it is that when selecting a new sequence number, processes work with so-called *safe* intervals from which they choose safe sequence numbers. A sequence number is called *safe* for process p if it can be used by process p as a tag without causing some other process' SC to succeed when it should fail. While using up this safe interval, the processes search for a new safe interval to use, and will jump to the next safe interval when one is found. The 'safe interval' is calculated *per process*; each process will test and identify safe intervals on its own, and may very well be using the same interval as another process. It is not the sequence number that is unique, but the (sequence number, pid)-pair.

We give a brief description of the simple selection algorithm here. A more elaborate description and correctness arguments can be found in [11].

```
    Constant: Δ = (2N + 1)N
    Constant: M = 'maxSeqnumtype' = 2^63 / N
    proc select(p) returns seqnumtype
10: a = A[procNum]
11: if ((a.pid = p) ∧ (a.seq ∈ [nextStart, nextStart ⊕ Δ)))
12:         nextStart = nextStart ⊕ Δ
13:         procNum = 0
14: else procNum = procNum + 1
15: if (procNum < N)
16:         val_p = val_p ⊕ 1
17: else val_p = nextStart
18:         nextStart = nextStart ⊕ Δ
19:         procNum = 0
20: return val_p
```

Fig. 10. The simple selection algorithm. Note: all $\oplus$ operations should be read as $\oplus_M$, where the $a \oplus_M b$ operation is defined as $a + b \bmod M$

The seqnumtype is ranged from $0 \ldots M-1$, and the safe intervals are chosen from this range. In this, the seqnumtype is considered to be circular: it is possible for an interval to start 'at the right' and ends 'at the left', because calculations are done with modulo M. The length of a safe interval is defined to be Δ (see figure 10). Each process p starts with a safe interval $[0, \Delta)$. The `select` function returns sequence numbers from this safe interval in order, but during each invocation of `select` it will also look for a next safe interval to use. This search happens as follows. First, a next interval to search is identified: initially $[nextStart, nextStart \oplus \Delta)$. Searching for the next safe interval is done by going through the announcement array `A`, inspecting one element of `A` each time `select` is called. First the element `A`$[procNum]$ is read in a (line 10), the current entry in `A` that is up for inspection. If the guard in line 11 is true, this means that the "tag" announced in `A` is in the interval that is

currently being inspected. This means that the interval $[nextStart, nextStart \oplus \Delta)$ is not safe to use, because another process has announced it is working with a value of X written by process p with a sequence number originating from the interval p is currently testing for safeness. The interval to inspect is changed to the next one and the *procNum* counter reset (lines 12, 13). If the guard in line 11 is false, *procNum* is incremented so the next entry in A can be inspected next time `select` is invoked (line 14). In line 15 the `select` algorithm checks if the whole interval has been inspected yet by comparing *procNum* to N. If this is not the case, the return value *val* is picked from the current safe interval (line 16). Otherwise it means that during the inspection of all entries in A, none of A's entries corresponded with a value from the interval that p currently inspected. In other words, the inspected interval is safe to use. Process p will change its safe interval to the safe interval it just identified, and immediately start looking for the next safe interval (lines 17-19).

An important aspect of this selection algorithm is that it requires only one operation on a shared variable: the read of A in line 10. This allows us to compress the selection algorithm into one single atomic statement, as we will see later.

4.2 Formal specification

We reuse the algorithm specification from the first proof, but with a few small changes concerning the data types. This is a result from the slightly different specification of `BitPid_LL`. We present the formal specification in figure 11.

```
Types:
   valuetype        (processtype, bit)
Ghost vars:
   start, ll, top   nat
   hist             array[nat] of valuetype
Private vars:
   v                valuetype
   arg              bit
   result           boolean

proc BitPidLLe(p)
   start := top
   choose ll with start ≤ ll ≤ top ; v := hist(ll)

proc BitPidSCe(p)
   if ll = top then
      top++ ; hist(top) := (p, arg) ; result := true
   else result := false end

proc BitPidVLe(p)
   result := (ll = top)
```

Fig. 11. Formal specifications for the bounded algorithm.

The correspondence between figure 3 and figure 11 is obvious. The only difference is the data type for `hist` and v: instead of a plain value, it now stores a $[pid, bit]$ pair. This way the "pid" LL/SC object can also return the process identifier of the process that stored the latest data value. This data type difference is reflected in the assignment to `hist(top)`, that now receives the value (p, arg) instead.

4.3 Rewriting the algorithm

Using the same rewriting rules from before we rewrite the algorithm to suit it to the new specification. This leads to the version of the program depicted in figure 13. We first discuss the declarations, printed in figure 12.

Types:

processtype	number from range $0 \ldots N-1$
tagtype	M-bit number, with $M = 63 - \log N$
valuetype	(tagtype, processtype, bit)

Variables:

Shared program vars:		
	`X`	valuetype
	`A`	**array**[processtype] **of** valuetype
Private program vars:		
	old, *chk*	valuetype
	tag, *nextStart*	tagtype
	procNum	processtype
	v, *tmp*	(processtype, bit)
	arg	bit
	result	boolean
Shared ghost vars:		
	`top`, `seq`	**int**
	`loc`	**array**[processtype, int] **of** int
	`tloc`	**array**[processtype, **int**] **of** tagtype
	`hist`	**array**[int] **of** (processtype, bit)
Private ghost vars:		
	start, *ll*, *oseq*, *cseq*, *jump*	int
	startSafe, *endSafe*	tagtype

Fig. 12. Declarations for the rewritten bounded algorithm

The ghost variables that we add are partly different. Apart from the specification vars *start*, *ll*, `top`, and `hist`, we reintroduce `loc` with the same function as before. Because the sequence number *seq* is not actually a sequence number anymore, but rather a number from a limited range (and not necessarily ordered in sequence), we rename it to the more intuitive *tag*. This allows us to introduce a ghost variable named `seq` that we can use to keep track of the *real* sequence numbers. In essence this ghost variable `seq` takes exactly the same role as the actual program variable *seq* from the first algorithm. A difference is the implicit meaning of this variable: instead of the '*seq.p* is the number of *p*'s *next* SC', now holds '`seq`.*p* is the number of *p*'s *last* SC'. Furthermore, because the LL procedure stores two local copies of `X` in this algorithm (in *old* and in *chk*), we want to be able to tell later on what these values are, independent of the algorithm. We add two more ghost variables, *oseq* and *cseq*, that are used to store a copy of `seq` at the time of reading `X`. This way the unique sequence number can be used to look up the value in the `hist` array to see which value has been obtained in *old* and *chk*. We have also added `tloc`, which can be considered `loc`'s little brother as its function is almost the same. However, instead of storing old `top` values, the purpose of `tloc` is to store old (non-unique) *tag* values. The variables *startSafe* and *endSafe* are used as markers to keep track of the beginning and the end of the *current* safe interval. The variable *jump* will store the number of jumps in the search for the next safe interval: if an inspected

interval is discarded as unsafe, the search algorithm 'jumps' to the next interval to inspect and the jump counter is increased.

Discussion In line 1 we start with adding the assignment *start* := `top`. The value returned by an LL is always (*old.pid*, *old.val*). Since *old* is assigned in line 1, we can set the return value *v* in line 1 already as well. We also set *oseq*, and assign *ll* with a value from the `loc` register, namely `loc`[*old.pid*, *oseq*] (note that this is equivalent to `loc`[`X`.*pid*, `seq`.(`X`.*pid*)]). Line 2 is unchanged, and in line 3 we add the assignment to *cseq* from `X`. Depending on what happened so far, it is also possible for a process to set a new *ll* value. Therefore we also add an if-statement to line 3, which possibly results in a reassigning *ll*. This is not an obvious choice, and is related to the second read of `X`. Before explaining this issue in more detail, we first finish the rest of the description of the modified algorithm. Since the return value has already been set in line 1, the original line 4 is discarded and the return happens in line 3.

In the SC procedure, we combine line 5 with line 6. The guards of the if-statements in line 5 and 6 are combined into one. This is allowed because in line 5 only local variables are checked. The merging of the lines is done by adding the negative guard from the CAS, rewritten using the CAS specification, to the guard of line 5. The CAS specification also leads to an else-branch for the new guard in which `X` and all ghost variables are updated. In line 7, we have removed the function call to `select` and replaced it with the `select` function itself. We are allowed to put the whole `select` algorithm from figure 10 into one single step because the procedure includes only one operation on a shared variable.

The assignment to *ll* in line 3 Our first version of the rewritten algorithm did not feature the second assignment to *ll*. By leaving this out there was an important aspect of the algorithm we did not cover in this "specification bridging" version. It concerns the purpose of the second read of `X` into *chk*. Recall that we wrote that at first glance the second read of `X` seems unnecessary. From the paper [2] that first introduced the algorithm that Jayanti and Petrovic base their bit/pid LL/SC object on we quote: "*tags are selected in such a way that a CAS [in the SC procedure] only succeeds if and only if no successful SC has occurred since the second read of `X` in p's LL operation*".

We can illustrate the different resulting possibilities as follows. Suppose that p is about to execute line 3. First, if *old* $\neq$ `X` (note that `X` is equivalent to *chk* here), then there will be no problems later on since a SC will fail. If *old* does equal `X`, there are two possibilities. The simple possibility is that nothing has happened to `X` while p went from line 1 to line 3: no SC's have taken place and it is still exactly the same as in line 1, and thus (*old* = *chk* = `X`).

Suppose that a process q last wrote `X` with $tag.q = \alpha$ and $arg.q = \mathcal{V}$. Process p reads `X` in line 1 such that $old.p = (\alpha, q, \mathcal{V})$, then goes 'dormant' without having performed line 2 yet, meaning that p's read has not yet been announced. In this while, q continues doing SC's and eventually arrives at the point where it reuses the tag α. This is possible because p did not announce yet and thus process q could see the interval containing α as safe. Process q continues and again writes $(\alpha, q, \mathcal{V})$ into `X`. Process p now continues its operation, writes `A`[p] in line 2 and reads `X` again in line 3. This is on appearance *exactly* the same `X` it read in line 1, and is in fact indistinguishable from it. Therefore it does not make a difference in operation if process p can legally continue working with this LL value and later (possibly) do a successful SC.

However, it *does* make a difference for the specification. The assignment to *ll* in line 1 is not correct anymore because *oseq* $\neq$ *cseq*. The algorithm has to change *ll*

```
        Functions:
        arc(a, b, c, m) returns bool =
            return (b ≤ a < b + c) ∨ (b ≤ a + m < b + c)

        calling(p)
50:         ( goto 1  ‖  goto 9  ‖  goto 10  ‖  choose arg , goto 5 )

        proc BitPid_LL(p)
1:          start := top; old := X; v := (X.pid, X.val);
            oseq := seq.(old.pid); ll := loc[old.pid, oseq];
2:          A[p] := (old.tag, old.pid, 0);
3:          chk := X; cseq := seq.(chk.pid);
            if (X = old) then ll := loc[chk.pid, cseq]; end if;
            goto 50;

        proc SC(p)
5:          if (old ≠ chk or old ≠ X) then result := false; goto 50;
            else X := (tag, p, arg); top++; hist(top) := (p, arg);
                seq++; loc[p, seq] := top; tloc[p, seq] := tag;
                result := true;
            end if;
7:          a := A[procNum];
            if a.pid = p ∧ arc(a.tag, nextStart, Δ, M) then
                nextStart := (nextStart + Δ) mod M; procNum := 0; jump++;
            else procNum++; end if
            if procNum < N then tag := (tag + 1) mod M; else tag := nextStart;
                nextStart := (nextStart + Δ) mod M; procNum := 0;
                jump := 0; startSafe := tag; endSafe := nextStart; end if
            goto 50;

        proc VL(p)
9:          result := (X = old = chk); goto 50;

        proc read(p)
10:         tmp := (X.pid, X.val); goto 50;
```

Fig. 13. The rewritten bounded algorithm

in this scenario, and ll will be set to $\mathtt{loc}[chk.pid, cseq]$, the chk counterpart of the value it got in line 1.

Initialization For the initialization of the variables, we pretend that process 0 performed a successful 'initializing SC' with an initial value $\mathcal{V}$. The resulting initial values for all processes p, numbers x, and bits b can be seen in figure 14.

4.4 Proof obligations

We can again formulate two proof obligations.

(Ob3.) $\quad v.r = \mathtt{hist}(ll.r) \ \wedge \ start.r \leq ll.r \leq \mathtt{top}$

(Ob4.) $\quad pc.r \in \{5, 9\} \ \Rightarrow \ (\mathtt{X} = old.r = chk.r) \equiv (ll.r = \mathtt{top})$

Of these, obligation (Ob3.) has to hold when exiting the LL procedure, and (Ob4.) when testing the validity of the current value, that is, in the SC and VL procedure.

Note that, since this proof is not the final version yet, not all of the invariants given here may be needed. They are all correct, though.

Shared program vars:
$\mathtt{X} = [0, 0, \mathcal{V}]$
$\mathtt{A}[p] = (0, 0, 0)$
Private program vars:
$old.p = chk.p = 0$
$tag.0 = procNum.0 = 1$
$\forall p : p \neq 0 : tag.p = procNum.p = 0$
$nextStart.p = \Delta$
$v.p = tmp.p = (0, \mathcal{V})$
$arg.p = \mathcal{V}$
$result.p = false$
Shared ghost vars:
$\mathtt{top} = 1$
$\mathtt{seq}.0 = 1$
$\forall p : p \neq 0 : \mathtt{seq}.p = 0$
$\mathtt{loc}[p, x] = x$
$\mathtt{tloc}[p, x] = 0$
$\mathtt{hist}(1) = \mathcal{V}$
Private ghost vars:
$start.p, ll.p, oseq.p, cseq.p = 0$

Fig. 14. Initialization for all variables in the bounded algorithm

Important notation note: In the proof we have to formulate invariants around the ghost variable `loc`. Very frequently we address either $\mathtt{loc}[\mathtt{X}.pid, \mathtt{seq}.(\mathtt{X}.pid)]$, $\mathtt{loc}[(old.r).pid, oseq.r]$, or $\mathtt{loc}[(chk.r).pid, cseq.r]$. These expressions are quite long. To increase readability we will therefore sometimes use the (syntactically incorrect) expressions $\mathtt{loc}[\mathtt{X}]$, $\mathtt{loc}[old.r]$, and $\mathtt{loc}[chk.r]$, respectively, while actually meaning their more elaborate counterparts.

4.5 Proof of correctness

General invariants We start with the value of `X` or, `hist(top)`. This leads to the obvious (Aq0). By definition, `hist(top)` is the value last stored by an SC in `X`.

(Aq0) $\quad \mathtt{hist}(\mathtt{top}) = \mathtt{X}$

We can also see that, as the `loc`-register holds old `top`-pointers again, the last `top`-pointer is stored in `X`'s `loc`-register. This is formulated in (Bq1).

(Bq1) $\quad \mathtt{top} = \mathtt{loc}[\mathtt{X}.pid, \mathtt{seq}.(\mathtt{X}.pid)]$

The `loc`-register behaves just like in the first algorithm. This means that it is an ordered list of increasing values. We formulate two invariants to express this, similar to (11.) and (12.) from the first proof but slightly different because of the shift in the `loc` register.

(Lq0) $\quad a \leq \mathtt{seq}.r \;\Rightarrow\; \mathtt{loc}[r, a] \leq \mathtt{top}$
(Lq1) $\quad a < b \;\wedge\; b \leq \mathtt{seq}.r \;\Rightarrow\; \mathtt{loc}[r, a] < \mathtt{loc}[r, b]$

The proofs for (Lq0) and (Lq1) are the same as (11.) and (12.).

The first obligation The first proof obligation consists of two parts, one about the value of v and one about the range of the chosen ll. We start with the range of ll. Just like before, we split this into two invariants:

(Cq1) $start.r \leq ll.r$
(Cq2) $ll.r \leq \texttt{top}$

Predicate (Cq1) is easy to prove. Assuming that the invariant is correct, the validity is threatened in line 1 and in line 3. In line 1, the value for $start.r$ and $ll.r$ are assigned. The assignments lead to $start.r = \texttt{top} \leq ll.r = \texttt{loc}[\texttt{X}.pid, \texttt{seq}.(\texttt{X}.pid)]$, but by (Bq1) we know that the top value equals this. In line 3 it's possible for $ll.r$ to change. However, its value changes once more to the current X value, which is (Bq1) equal to top. We also know that $start.r$ is always smaller than top, something that we express in the following invariant.

(Dq1) $start.r \leq \texttt{top}$

The validity of predicate (Dq1) is obvious, and this expression is enough to prove the validity of (Cq1).

The second part, (Cq2), is even easier. We want to show that the value for $ll.r$ is smaller than top at any point. Both in line 1 and in line 3 this is true because of (Bq1).

We continue with finding the value of v. We first express the invariant:

(Bq3) $v.r = \texttt{hist}(ll.r)$

This invariant is threatened in 3 places: in line 1, line 3 and line 5 changes are being made to the variables of (Bq3). We look at them in order of increased complexity, first line 1, then line 5 and finally line 3.

In line 1, the values of $v.r$ and $ll.r$ are set. First note that $ll.r = \texttt{loc}[\texttt{X}.pid, \texttt{seq}.(\texttt{X}.pid)]$ which, according to (Bq1), equals top. Since (Aq0) said that this is equal to X, and $v.r$ is set to X, validity of (Bq3) is preserved in line 1.

In line 5, hist is changed. To be more specific, the value changed is hist(top) after incrementing top. With (Cq2) we showed that $ll.r \leq \texttt{top}$ and thus line 5 can not change $\texttt{hist}(ll.r)$.

In line 3, $ll.r$ may be changed. We need to show that whenever that is the case, the changed $ll.r$ still satisfies (Bq3). The first observation we can make is that this is only a threat if process r itself executes line 3 when $(\texttt{X} = old.r)$. Assuming that this is the case, $ll.r$ will receive a new value. First notice something about v: it is appointed in line 1, copied directly from X, in the same line that old is assigned also directly from X. We make the correspondence between $v.r$ and $old.r$ explicit in the next predicate.

(Bq4) $v.r = ((old.r).pid, (old.r).val)$

This allows us to read (Bq3) as $((old.r).pid, (old.r).val) = \texttt{hist}(ll.r)$. Because $(old.r).val = \texttt{X}.val$ when making the change to $ll.r$ in line 3, (Aq0) and (Bq1) can be used to prove that (Bq3) is valid here.

The second obligation The second proof obligation is the part about the correspondence between the tests in the specification and the tests in the algorithm in the SC and VL operation. We repeat the obligation here.

(Ob4.) $pc.r \in \{5, 9\} \;\Rightarrow\; (\texttt{X} = old.r = chk.r) \equiv (ll.r = \texttt{top})$

We split the proof, leading to:

(Zq0) $pc.r \in \{5, 9\} \;\wedge\; (\texttt{X} = old.r = chk.r) \;\Rightarrow\; (ll.r = \texttt{top})$
(Zq1) $pc.r \in \{5, 9\} \;\wedge\; (ll.r = \texttt{top}) \;\Rightarrow\; (\texttt{X} = old.r = chk.r)$

Our approach is showing equivalence for all record fields under these conditions. For example, (Zq0) can be read as:

$$pc.r \in \{5,9\} \ \wedge \ \begin{pmatrix} \mathtt{X}.tag \\ \mathtt{X}.pid \\ \mathtt{X}.val \end{pmatrix} = \begin{pmatrix} old.tag \\ old.pid \\ old.val \end{pmatrix} = \begin{pmatrix} chk.tag \\ chk.pid \\ chk.val \end{pmatrix} \ \Rightarrow \ (ll.r = \mathtt{top})$$

We need explicit expressions for the values of the record fields before we can say anything meaningful about (Zq0) and (Zq1).

Expression for the *val*-field The easiest to prove seems to be the value field, so we start with formulating (Fq0).

(Fq0) $(old.r).val = \mathtt{hist}(\mathtt{loc}[(old.r).pid, oseq.r]).val \ \wedge$
$(chk.r).val = \mathtt{hist}(\mathtt{loc}[(chk.r).pid, cseq.r]).val$

In line 1 and 3, the place where *old* and *chk* are set, we can use (Aq0) and (Bq1) to prove that these are indeed the right values. Validity is also threatened in line 5 because of the changes to `hist` and `loc` there. This can never be a real threat: the `hist` value changed there comes later than the `hist` values from (Fq0), and the same goes for the `loc` value. This information hasn't been put in an invariant yet, so we formulate the following two expressions:

(Lq2) $\mathtt{loc}[(old.r).pid, oseq.r] \leq \mathtt{top} \ \wedge \ \mathtt{loc}[(chk.r).pid, cseq.r] \leq \mathtt{top}$
(Lq4) $oseq.r \leq \mathtt{seq}.((old.r).pid) \ \wedge \ cseq.r \leq \mathtt{seq}.((chk.r).pid)$

The first of these, (Lq2), a somewhat obvious invariant, can be used to show that the `loc` values from *old* and *chk* are at most `top`. Validity is threatened in line 1 and 3 (when setting *old* and *chk*), but (Bq1) reveals that the values assigned are safe. The second, (Lq4), simply states that *old* and *chk* sequence numbers are always smaller than the current sequence number of their process. The proof of this is obvious.

Predicates (Lq2) and (Lq4) are enough to show the validity of (Fq0) in line 5, completing the proof for (Fq0).

Expression for the *tag*-field The only way we can say something about the *tag* values is by looking at the `tloc` register. The most obvious information that `tloc` holds is the *tag* that is currently used by `X`. We formulate this as (Gq1).

(Gq1) $\mathtt{tloc}[\mathtt{X}.pid, \mathtt{seq}.(\mathtt{X}.pid)] = \mathtt{X}.tag$

Then we can continue with expressions for the *tags* of *old* and *chk*. It should be clear that they can be expressed like (Gq2).

(Gq2) $\mathtt{tloc}[(old.r).pid, oseq.r] = (old.r).tag \ \wedge$
$\mathtt{tloc}[(chk.r).pid, cseq.r] = (chk.r).tag$

To prove this, (Gq1) can be used in lines 1 and 3 to show that when assigning *old* and *chk* these values are correct. In line 5, it is because of (Lq4) that (Gq2) is not invalidated.

Expression for *ll* The last variable occurring in (Zq0) and (Zq1) that we have not explicitly expressed yet is *ll*. The value for *ll.r* is first set in line 1 and is therefore equal to *old.r*, leading to the following expression.

(Cq4) $pc.r \in \{2,3\} \ \Rightarrow \ ll.r = \mathtt{loc}[(old.r).pid, oseq.r]$

Outside of lines 2 and 3 this is not necessarily valid because $ll.r$ might change in line 3. The validity of (Cq4) is threatened in line 5's operation on `loc` but we can use (Lq4) to show it doesn't affect the value in this invariant.

The value for $ll.r$ can also be set in line 3. This will only happen if the guard in line 3 is evaluated to true or in other words, only when `X` equals $old.r$. But while executing line 3, `X` is equivalent to $chk.r$ as well. Using this knowledge we create another expression for $ll.r$ valid outside the LL procedure as well.

$$\text{(Cq3)} \qquad \begin{array}{l} ll.r = \texttt{loc}[(old.r).pid, oseq.r] \vee \\ (ll.r = \texttt{loc}[(chk.r).pid, cseq.r] \ \wedge \ chk.r = old.r) \end{array}$$

We can see this must be true. The first disjunct covers the line 1 ll alternative, the second disjunct covers the line 3 ll alternative. We can use (Cq4) to show that if ll does *not* get a new value in line 3, the first disjunct of (Cq3) was already true and validity is not threatened. And similar to (Cq4), the threat to (Cq3) in line 5 is eliminated by (Lq4).

Just as (Cq3) was a stronger version of (Cq4), we can make the expression for $ll.r$ even stronger by adding the information about chk and old to the first disjunct of (Cq3) as well, i.e. the fact that $chk.r \neq old.r$ in that case. However, that claim is not entirely true: during the LL operation itself this would not make much sense. Therefore we add the demand that this only holds outside of the LL procedure. This is formulated in the next invariant.

$$\text{(Cq5)} \qquad \begin{array}{l} pc.r \notin \{1,2,3\} \ \Rightarrow \ (ll.r = \texttt{loc}[(old.r).pid, oseq.r] \ \wedge \ chk.r \neq old.r) \vee \\ \qquad\qquad\qquad\qquad (ll.r = \texttt{loc}[(chk.r).pid, cseq.r] \ \wedge \ chk.r = old.r) \end{array}$$

When exiting line 3, predicate (Cq4) gives the required information for the proof that from then on, (Cq5) will hold and like the previous invariants (Lq4) aids in the proof of validity of (Cq5) in line 5.

Proving (Zq1) With the invariants formulated so far, we can make a few advances. The expression $(ll.r = \texttt{top})$ can be transformed using (Cq5) and (Bq1) into:

$$\begin{array}{l} ((\texttt{loc}[old.r] = \texttt{loc}[\texttt{X}] \ \wedge \ chk.r \neq old.r) \vee \\ (\texttt{loc}[chk.r] = \texttt{loc}[\texttt{X}] \ \wedge \ chk.r = old.r)) \end{array}$$

If we use this expression to replace the part $(ll.r = \texttt{top})$ in (Zq1), this leads to:

$$\text{(*)} \qquad \begin{array}{l} pc.r \notin \{1,2,3\} \ \wedge \ ((\texttt{loc}[old.r] = \texttt{loc}[\texttt{X}] \ \wedge \ chk.r \neq old.r) \vee \\ \qquad\qquad\qquad\qquad (\texttt{loc}[chk.r] = \texttt{loc}[\texttt{X}] \ \wedge \ chk.r = old.r)) \\ \quad \Rightarrow \ (\texttt{X} = old.r = chk.r) \end{array}$$

(The $pc.r \in \{5, 9\}$ has been extended to cover *everything* outside of the LL procedure). This means we have to prove both disjuncts. We first formulate the following predicate that, when proved valid, should ensure the validity of (*) in case of the second disjunct.

$$\text{(TTq0)} \qquad \texttt{loc}[chk.r] = \texttt{loc}[\texttt{X}] \ \wedge \ chk.r = old.r \ \Rightarrow \ chk.r = \texttt{X}$$

The easiest way to show that (TTq0) is valid is by showing that this equality holds for each record field of the data type: *tag*, *pid*, and *val*.

First, $\texttt{loc}[chk.r] = \texttt{loc}[\texttt{X}] \ \wedge \ chk.r = old.r \ \Rightarrow \ (chk.r).tag = \texttt{X}.tag$. We can use (Gq1) and (Gq2) to replace the *tag* expressions with `tloc` expressions. This leads to: $\texttt{loc}[chk.r] = \texttt{loc}[\texttt{X}] \ \wedge \ chk.r = old.r \ \Rightarrow \ \texttt{tloc}[\texttt{X}] = \texttt{tloc}[chk.r]$. It is already known that `loc` values are unique (`loc` stores `top` values and every `top` value is only stored once at one place). Therefore, if the `loc` values are equal, this means that the indices of the `loc` pointing to this value must be equal as well, and

in this expression the indices used for the tloc's are the same as the indices used for the loc's.

We can formalize this in an expression (Lq5), of which the validity through line 5 is ensured by (Lq0).

$$\text{(Lq5)} \qquad (\texttt{loc}[r,a] = \texttt{loc}[q,b] \ \wedge \ a \leq \texttt{seq}.r \ \wedge \ b \leq \texttt{seq}.q) \equiv (r = q \ \wedge \ a = b \ \wedge \ a \leq \texttt{seq}.r)$$

This predicate can be used because of (Lq4) that states that *cseq.r* satisfies the restrictions of (Lq5)'s "*a*".

Second, $\texttt{loc}[chk.r] = \texttt{loc}[\texttt{X}] \ \wedge \ chk.r = old.r \ \Rightarrow \ (chk.r).pid = \texttt{X}.pid$. This is true also, because by the same reasoning as the *tag* field, the indices of loc in this expression are the same and thus the *pid*'s are the same as well.

Third and last, $\texttt{loc}[chk.r] = \texttt{loc}[\texttt{X}] \ \wedge \ chk.r = old.r \ \Rightarrow \ (chk.r).val = \texttt{X}.val$. Using (Aq0) and (Bq1) the term $\texttt{X}.val$ can be seen as $\texttt{hist}(\texttt{loc}[\texttt{X}])$. Then, by applying the equality relation from (TTq0)'s antecedent, we are able to write this as $\texttt{hist}(\texttt{loc}[chk.r])$ and according to (Fq0), this is the expression for $(chk.r).val$.

This concludes the proof of (TTq0).

We continue with a proof for the first disjunct of (*). This can be proved if we can show that in this case, $\texttt{loc}[\texttt{old}.r] \neq \texttt{top}$. We formulate this as predicate (Lq6).

$$\text{(Lq6)} \qquad pc.r \notin \{1,2,3\} \ \wedge \ old.r \neq chk.r \ \Rightarrow \ \texttt{loc}[old.r] < \texttt{top}$$

An inequality relation would be enough in (Lq6), but since loc values can not be larger than top we can use a less-than relation.

Intuitively, we know that (outside of the LL procedure) the loc value referred to by $old.r$ is at most the loc value referred to by $chk.r$. and furthermore, if $old.r \neq chk.r$ that their loc values can not be the same. Also, since $chk.r$ is set in line 3, its loc value is at most X's loc value. We formulate these thoughts in the following three invariants.

$$\text{(Lq7)} \qquad pc.r \notin \{1,2,3\} \ \Rightarrow \ \texttt{loc}[old.r] \leq \texttt{loc}[chk.r]$$
$$\text{(Sq5)} \qquad old.r \neq chk.r \ \Rightarrow \ \texttt{loc}[old.r] \neq \texttt{loc}[chk.r]$$
$$\text{(Sq6)} \qquad pc.r \notin \{1,2,3\} \ \Rightarrow \ \texttt{loc}[chk.r] \leq \texttt{loc}[\texttt{X}]$$

By putting these three invariants together with (Bq1), predicate (Lq6) can be proved in the following way:

$$pc.r \notin \{1,2,3\} \ \wedge \ old.r \neq chk.r \ \Rightarrow \ \texttt{loc}[old.r] < \texttt{loc}[chk.r] \leq \texttt{loc}[\texttt{X}] = \texttt{top}$$

That leaves us with these three invariants to prove.

First (Sq6), which is made true in line 3. It is threatened in line 5 when new loc values are written but this can be shown harmless with (Lq2).

Next, (Lq7). To know that (Lq7) indeed holds after exiting the LL procedure, we look at line 3. From (Lq2) and (Bq1) it follows that $\texttt{loc}[old.r] \leq \texttt{loc}[\texttt{X}]$. Since while executing line 3 $chk.r$ can be interchanged with X, $\texttt{loc}[old.r] \leq \texttt{loc}[chk.r]$ holds after the LL. The threat of overwriting these loc values in line 5 is eliminated by invariant (Lq4).

Finally, (Sq5). The inequality $old.r \neq chk.r$ can hold in three ways, one of each for the record fields. We split the proof into three sub-expressions. Together, if valid, they prove (Sq5)'s validity.

$$\text{(sub1)} \qquad (old.r).tag \neq (chk.r).tag \ \Rightarrow \ \texttt{loc}[old.r] \neq \texttt{loc}[chk.r]$$
$$\text{(sub2)} \qquad (old.r).pid \neq (chk.r).pid \ \Rightarrow \ \texttt{loc}[old.r] \neq \texttt{loc}[chk.r]$$
$$\text{(sub3)} \qquad (old.r).val \neq (chk.r).val \ \Rightarrow \ \texttt{loc}[old.r] \neq \texttt{loc}[chk.r]$$

We discuss the predicates in the order (sub2), (sub3) and (sub1). Remember that $\texttt{loc}[old.r] \neq \texttt{loc}[chk.r]$ is an abbreviation for $\texttt{loc}[(old.r).pid, oseq.r] \neq \texttt{loc}[(chk.r).pid, cseq.r]$.

(sub2): The *pid*'s of *old.r* and *chk.r* differ. Near (Lq5) we already explained that `loc` values are unique. Since the index of the value is different, it follows that the values should be different as well. This follows from (Lq5), because (Lq4) provides enough information to prove that the values lie within range.

(sub3): The *val*'s can be translated using (Fq0). This leads to $\texttt{hist}(\texttt{loc}[old.r]) \neq \texttt{hist}(\texttt{loc}[chk.r])$, and this implies that (sub3) is valid.

(sub1): To prove this invariant, we start by replacing the *tag* fields with (Gq2). Thus, $\texttt{tloc}[(old.r).pid, oseq.r] \neq \texttt{tloc}[(chk.r).pid, cseq.r]$ should lead to the inequality in the `loc` values we want, so $(old.r).pid \neq (chk.r).pid \vee oseq.r \neq cseq.r \Rightarrow \texttt{loc}[(old.r).pid, oseq.r] \neq \texttt{loc}[(chk.r).pid, cseq.r]$. Using (Lq5) and (Lq4) again, it is easily shown that this is true.

To retrace our steps: the sub-expressions proved (Sq6) which was the final key in proving (Sq5)'s validity, required to complete the proof of (Lq6). Along with (TTq0), (Lq6) proves (Zq1), the first half of the second proof obligation (Ob4.).

Proving (Zq0) The more difficult part of the second proof obligation is the proof of (Zq0), reprinted below.

$$\text{(Zq0)} \qquad pc.r \in \{5,9\} \ \wedge \ (\texttt{X} = old.r = chk.r) \ \Rightarrow \ (ll.r = \texttt{top})$$

The proof of (Zq0) has not been completed: it is the only part of the proof of the second algorithm that we did not manage to verify. What follows here is the reasoning we followed in our proof attempt, and how this reasoning leads to certain assumed invariants that are needed to complete the proof.

Using (Cq5) and (Bq1), we can rewrite (Zq0) to:

$$pc.r \in \{5,9\} \ \wedge \ (\texttt{X} = old.r = chk.r) \ \Rightarrow \\ \texttt{loc}[(chk.r).pid, cseq.r] = \texttt{loc}[\texttt{X}.pid, seq.(\texttt{X}.pid)]$$

From this expression we can see that the validity of (Zq0) depends on the value of *cseq.r* in the following way:

$$\text{(Cq7)} \qquad pc.r \notin \{1,2,3\} \ \wedge \ (\texttt{X} = old.r = chk.r) \ \Rightarrow \ cseq.r = \texttt{seq}.(\texttt{X}.pid)$$

If we can show that (Cq7) holds, this means that (Zq0) is indeed a valid invariant. This is not an easy task because the selection of tags is such an intricate process.

The only threatening program step to (Cq7)'s validity is line 5, where changes can be made to `X` or `seq`. The conditional of the implication consists of three parts: $pc.r \notin \{1,2,3\}$, $old.r = chk.r$, and $\texttt{X} = chk.r$. The first two parts can not change with an execution of line 5 and are irrelevant. This leaves the part $\texttt{X} = chk.r$. There are two possibilities: either $\texttt{X} = chk.r$ was true prior to the execution of line 5, and thus $cseq.r = \texttt{seq}.(\texttt{X}.pid)$ was also true; or $\texttt{X} = chk.r$ was false prior to the execution of line 5, in which case you don't know whether $cseq.r = \texttt{seq}.(\texttt{X}.pid)$ was true or not.

Suppose $\texttt{X} = chk.r$. If any process p writes a new `X`, it will invalidate this part of the conditional. The reason for this is that the new `X` will be (tag, p, arg). If process p is $\texttt{X}.pid$ (required if $\texttt{X} = chk.r$ should remain true), this means that the *tag* should be the same as well, but the *tag* is chosen in such a way that this is not possible. Thus, after completing line 5 the statement $\texttt{X} = chk.r$ is not true anymore,

the conditional is made false and the implication true again. We made a nontrivial claim about the selected tags here, that follows from the next predicate.

(W6b) $pc.(\mathtt{X}.pid) \neq 7 \;\Rightarrow\; \mathtt{X}.tag \notin arc[tag.(\mathtt{X}.pid), endSafe.(\mathtt{X}.pid)]$

We give the proof of this predicate later, after completing the discussion of (Cq7)'s validity through line 5.

Suppose $\mathtt{X} \neq chk.r$. The question is whether the conditional remains false, or whether the conditional is made true along with the rest of the implication. It should be clear that the only process for which line 5 is a problem is process $p = (old.r).pid$. Now, the conditional is made true iff $tag.p = (chk.r).tag$. Intuitively, we say that a *tag* is *guaranteed* by the algorithm to be safe, and thus different from $(chk.r).tag$. Therefore we require, if $pc.r \notin \{1,2,3\}$ and $\mathtt{X} \neq chk.r$, that it will never be the case that $\mathtt{X} = chk.r$. The only unique part that influences this is the tag, so we express this as follows:

(W1b) $pc.r \notin \{1,2,3\} \;\wedge\; \mathtt{X} \neq chk.r \;\wedge\; chk.r = old.r \;\wedge$
$p = (chk.r).pid \;\wedge\; pc.p \neq 7 \;\wedge \quad\Rightarrow\quad tag.p \neq (chk.r).tag$

Note that the addition of the extra parts in the conditional are not an issue: they are true in this application of the predicate, and help in proving (W1b) later on.

With this, the proof of (Cq7) can be completed. However, without a proof of correctness for the new predicates we introduced, (W6b) and (W1b), this proof is still incomplete. This is what we concentrate on now.

Claims about *tag* Both (W6b) and (W1b) are statements expressing information about *tag* in different situations. The selection of new *tag* values is the part of the algorithm that makes it most complicated, especially since the search for new safe intervals to choose *tag* from is spread over multiple invocations of the selection procedure.

This part of the proof has not been finished yet, so we write this part under the assumption that the following claims are true.

(W1) $pc.r \notin \{1,2,3\} \;\wedge\; old.r = chk.r \;\wedge\; p = (old.r).pid \;\Rightarrow$
$(old.r).tag \notin arc[tag.p \oplus (\#pc.p = 7), endSafe.p]$

(W9) $pc.r = 7 \;\wedge\; (guard_7.r \;\vee\; (\neg guard_7.r \;\wedge\; procNum.r + 1 < N)) \;\Rightarrow$
$tag.r \oplus 1$ '$<$' $endSafe.r$

(W4short) $jump.r < 2N + 1$

Predicate (W1) poses a restriction on the selected tag compared to the tag from $old.r$. Predicate (W9) claims that, under the conditions when a new tag is selected from the current safe interval, the end of the safe interval has not been reached yet. Predicate (W4short) is used to limit the amount of 'jumps', a jump is performed every time an inspected interval is found to be unsafe and a jump to a next inspection interval is made.

Using these assumptions, we can prove (W1b) relatively easy: it follows directly from (W1). Since the conditional of (W1) is mostly the same as the one from (W1b), and in addition $pc.p \neq 7$, (W1) can be used to claim $(old.r).tag \notin arc[tag.p, endSafe.p]$, and if it is not in that arc, it can not be equal to the first element in that arc or $(old.r).tag \neq tag.p$; exactly what we wanted to know.

Predicate (W6b) is more complicated. We reprint it here, along with a complementary predicate (W6a) that covers the part outside (W6b)'s range. The proof of (W6a) is trivial.

(W6a) $pc.(\mathtt{X}.pid) = 7 \;\Rightarrow\; \mathtt{X}.tag = tag.(\mathtt{X}.pid)$
(W6b) $pc.(\mathtt{X}.pid) \neq 7 \;\Rightarrow\; \mathtt{X}.tag \notin arc[tag.(\mathtt{X}.pid), endSafe.(\mathtt{X}.pid)]$

The threat to the validity (W6b) occurs in line 7. First, note that this is only an issue if it is executed by process $p = \mathtt{X}.pid$. The newly chosen tag is either the old tag incremented by one (in case of selecting a new one from the current safe interval) or *nextStart* (in case of starting from a new safe interval). We distinguish between these two cases. In the first case, predicate (W9) can be used: it shows that $tag.r \oplus 1$ has not reached the end of the current safe interval yet. And since $tag.(\mathtt{X}.pid)$ can (here) be interchanged with $\mathtt{X}.tag$, we know the new situation is still valid. In the second case, we need to make sure that the that the new safe interval does not contain the tag that is currently in X. For this, we formulate:

(Nsq5) $nextStart.r \notin arc[startSafe.r - \Delta, endSafe.r]$
(W8) $tag.r \notin arc[startSafe.r, endSafe.r]$

These predicates are enough to prove it if we use them with process $p = \mathtt{X}.pid$. This can be seen as follows:

{ (W8) } $tag.\mathtt{X}.pid \notin arc[startSafe.\mathtt{X}.pid, endSafe.\mathtt{X}.pid]$
{ (Nsq5) } $nextStart.\mathtt{X}.pid = tag.\mathtt{X}.pid \notin arc[startSafe.\mathtt{X}.pid - \Delta, endSafe.p]$

Since the new *tag* becomes *nextStart* and $\mathtt{X}.tag$ can be interchanged with $tag.(\mathtt{X}.pid)$ we can, using reasoning about the location in the arc, conclude that in the new situation, $\mathtt{X}.tag$ is indeed not in the arc described in (W6b).

Once again we have managed to create two extra predicates that need to be proved. We start with (W8), that can be read as "process r's current tag comes from r's current safe interval". The validity of (W8) is threatened in line 7 with the selection of a new tag (and possibly new safe interval). If the selected tag is the start of a new safe interval, (W8) is valid because *startSafe* and *endSafe* change accordingly. If the new tag is coming from the current safe interval, we can use assumption (W9) to guarantee the end of the safe interval has not been reached yet and the validity of (W8) is maintained.

Predicate (Nsq5) is another claim of which the description sounds easy: the (possible) new safe interval should never be allowed to overlap with the current safe interval, and therefore *nextStart* is not allowed to get in the range between $startSafe.r - \Delta$ and $endSafe.r$. We express the value of *nextStart* in the obvious invariant (W7).

(W7) $nextStart.r = endSafe.r \oplus \Delta \cdot jump.r$

The amount of jumps is limited (see assumption (W4short)) to at most $2N$, so we can make a claim about the value of $\Delta jump.r$:

(W2) $jump.r \cdot \Delta < M - \Delta$

Under the restrictions of the different variables, we can show that (W2) is correct using the following equational reasoning:

{ (W4short) }
$jump < 2N + 1 \;\Rightarrow\; jump \cdot \Delta < (2N + 1)\Delta \;\Rightarrow$
{ Using $\Delta = (2N + 1)N$ }
$jump \cdot \Delta < N(2N + 1)^2$
Also:
$M - \Delta = M - (2N + 1)N$

{ Since $N < 2^{15}$, suppose $N = 2^{15}$. Then we can change these expressions to }
$jump \cdot \Delta < 2^{47} + 2^{32} + 2^{15}$
$M - \Delta = 2^{48} - 2^{31} - 2^{15}$
And thus, $jump \cdot \Delta < M - \Delta$

The invariants (W7) and (W2) are enough to prove (Nsq5). We can read (Nsq5) as $nextStart.r \notin arc[(endSafe.r - 2\Delta) \mathbf{mod}\ M, endSafe.r]$. Consider *nextStart*, by (W7) equal to $(endSafe + \Delta \cdot jump) \mathbf{mod}\ M$. From (W2) we know that the part $\Delta \cdot jump$ herein is less than $M - \Delta$. Then, because of the circular length M nature of the interval, this means *nextStart* can never be in the range specified by (Nsq5), and the proof of (Nsq5) is completed.

4.6 Observations

Unfinished proof Despite the effort put into completing the second part of the proof, we haven't been able to do so. The major problem was proving the correctness of the part in which a new tag is selected. This is a process more complex than one would suspect at first sight. In this section we discuss in more detail the invariants that we have used to complete the proof, but that we were not able to prove.

(W4short) $\quad jump.r < 2N + 1$

Possibly the most complicated expression to prove is (W4short). It is a delimiter for the amount of jumps. The algorithm should indeed guarantee that a search for a next safe interval is completed within a limited amount of jumps, such that the end of the current safe interval is not exceeded. Since the length of an interval is $\Delta = (2N + 1)N$ the amount of jumps is limited to $2N + 1$. This predicate possibly follows from a more complicated and detailed version expressing the same information:

$$
\begin{aligned}
(\text{W4}) \quad & \#\{i \in arc[nextStart.r, startSafe.r] \mid \\
& \quad (\mathtt{X}.pid = r \ \wedge\ \mathtt{X}.tag = i) \vee \\
& \quad (\exists q : (A[q].pid = r \ \wedge\ A[q].tag = i) \vee \\
& \qquad (pc.q = 2 \ \wedge\ (old.q).pid = r \ \wedge\ (old.q).tag = i)) \\
& \} + jump.r \leq 2N
\end{aligned}
$$

This motivation behind this expression is the following: the range $arc[nextStart.r, startSafe.r]$ is the part of the circular tag space that remains to choose a safe interval from. The number of tags in this region that are currently 'held' by other processes, either in X, or in the announcement array, or still *awaiting* to be announced (that is, for processes that are currently at line 2) is limited to a maximum of N since there are N processes. Every time a jump occurs, it means that the range $arc[nextStart.r, startSafe.r]$ is made smaller and *at least* one unsafe entry has been discarded, and also that the counter *jump* is incremented.

$$
\begin{aligned}
(\text{W1}) \quad & pc.r \notin \{1,2,3\} \ \wedge\ old.r = chk.r \ \wedge\ p = (old.r).pid \ \Rightarrow \\
& \quad (old.r).tag \notin arc[tag.p \oplus (\#pc.p = 7), endSafe.p]
\end{aligned}
$$

The purpose of (W1) is essentially to serve as a base expression to formulate (W1b), the invariant that tells us that the *tag* of a process is, under certain conditions, different from another process's *chk.tag* value. Put in words, (W1) claims "For a process r that has completed its LL operation and for which the *old* and *chk* values are equal, the *tag* field of *old.r* is *not* in the remaining safe interval of the process from the *pid* field of *old.r*". The addition of the '$\oplus(\#pc.p = 7)$' comes from the fact that this process may not have updated its *tag* value yet.

(W9) $pc.r = 7 \wedge (guard_7.r \vee (\neg guard_7.r \wedge procNum.r + 1 < N)) \Rightarrow$
$tag.r \oplus 1$ '<' $endSafe.r$

With (W9) we express the condition that the end of the current safe interval is not reached yet at the moment you choose the next entry in the current safe interval. This seems hard to verify, but we suspect this could be done by looking at $(tag.r - startSafe.r)$ **mod** M and restricting this to a maximum value of Δ.

5 The prover guide PVS

As stated in the introduction, we relied heavily on the proof assistant PVS[3] while constructing the proof. PVS provides "an integrated environment for the development and analysis of formal specifications, and supports a wider range of activities involved in creating, analyzing, modifying, managing, and documenting theories and proofs."[15]. In short, it offers us an environment in which we can construct and verify the proof of correctness for the algorithm. We use this section to give some background about how we used PVS. The examples given in this chapter only involve the first (unbounded sequence number) algorithm, unless otherwise stated.

5.1 Environment and algorithm

We model the algorithm in PVS as a state space consisting of the current state (values) of all different variables. We can then look at the system as a whole. In figure 15 we have depicted a part of the type definitions from the PVS file.

```
Item: TYPE+
N: posnat
Process: TYPE = below[N]
Bit: TYPE = below[2]
Tagtype: TYPE = [Process, int]

state: TYPE = [#
   hist: [int -> Item] ,
   top: int ,
   val: [[Process, Bit] -> Item] ,
   loc: [Tagtype -> int] ,
   ...
#]
```

Fig. 15. A part of the PVS type definitions for the first proof

In the definitions, the data type `state` is used to reflect the current state space and contains all variables, represented as record entries in a `state`-type variable. Some variables from the pseudo-code are not needed for us. For example, we have not added q or k in the definition of the state.They are not needed because they are private variables that are only used to store the separate fields of another private variable *tag*. In PVS, we avoid using them and access *tag*'s record fields directly instead. For a process r in a state x, q and k can are referenced using `x`tag(r)`1` and `x`tag(r)`2`, respectively.

Executions of atomic steps in the algorithm can be defined by giving the state space changes that result from these steps. In other words, using this data type we

[3] PVS (Prototype Verification System), version 3.2

can outline our algorithm by defining the changes in the state space each atomic step makes.

To illustrate this with an example we have given the translation of line 8 of the algorithm to PVS in figure 16. It is a straightforward translation of (pseudo-)code to the state space changes. The line `step8(p, x, y): bool` is the declaration: a process `p`, an initial/current state `x` and a new state `y` define the function `step8` that returns a boolean value. This 'return value' is *true* iff the variables p, x and y satisfy the requirements for line 8. These requirements are in the body of the `step8` function. First, the *pc* of process p in the current state x equals 8. Second, the state changes should reflect what happens in line 8. There are two possible states that can result from performing line 8. The result depends on the guard of the if-statement. We have to keep in mind that the changes in the variables do not happen sequentially, instead the whole state with all variables is changed. This results in a difference between for example the (sequential) lines `top++; hist(top)` := *arg*, and the parallel state change `y = x WITH [ ...'top := x'top + 1, 'hist(x'top + 1) := x'arg(p) ...]`.

```
8:      if (x = tag) then
           x := [p, seq];
           top++;
           hist(top) := arg;
           loc[p, seq] := top;
           seq++;
           result := true;
        else
           result := false;
           goto 50;
        end if;
```

```
step8(p, x, y): bool =
   x'pc(p) = 8 AND
   IF x'xshared = x'tag(p) THEN
      y = x WITH [
         'xshared := (p, x'seq(p)) ,
         'top := x'top + 1 ,
         'hist(x'top + 1) := x'arg(p) ,
         'loc(p,x'seq(p)) := x'top + 1 ,
         'seq(p) := x'seq(p) + 1 ,
         'result(p) := true ,
         'pc(p) := 9
      ]
   ELSE
      y = x WITH [
         'result(p) := false ,
         'pc(p) := 50
      ]
   ENDIF
```

Fig. 16. Line 8 in the algorithm (left) and its translation to PVS (right)

All other program steps have been built similarly.

Since our proof is constructed around invariants that should retain their validity under *any* program step (i.e. under any change in the state space) we also define an action that corresponds with '*any step by any process*'. We do this by creating a new function `step(p, x, y): bool`, that consists of a disjunction of all defined program steps.

Note on translating line 7 of the bounded algorithm Line 7 of the second algorithm is a bit tricky to translate to PVS. The PVS model works by using states, and all variable changes in a single program step are done 'simultaneously'. But in line 7, the changes in some variables depend on tests of other variables that may have changed in the same line. We discuss the translation here.

The new values of most variables depend on the results of both if-statements. For readability reasons, we shorten the first if-statement to '$guard_7$'. Then, if we rewrite the assignments using only the initial values for the different variables, we can see that:

$$nextStart := \begin{cases} nextStart + \Delta \bmod M & ; guard_7 \\ nextStart + \Delta \bmod M & ; \neg guard_7 \ \wedge \ procNum + 1 = N \\ \text{unchanged} & ; \text{otherwise} \end{cases}$$

$$procNum := \begin{cases} 0 & ; guard_7 \vee \neg guard_7 \wedge procNum + 1 = N \\ procNum + 1 & ; \neg guard_7 \end{cases}$$

$$tag := \begin{cases} tag + 1 \textbf{mod } M & ; guard_7 \vee (\neg guard_7 \wedge procNum + 1 < N) \\ nextStart & ; \neg guard_7 \wedge procNum + 1 = N \end{cases}$$

$$jump := \begin{cases} jump + 1 & ; guard_7 \\ 0 & ; \neg guard_7 \wedge procNum + 1 = N \\ \text{unchanged} & ; \text{otherwise} \end{cases}$$

This is what happens to the variables (we have omitted the assignments to *startSafe* and *endSafe*). In PVS, the complete line 7 is expressed as follows:

```
step7(p,x,y): bool =
  x'pc(p) = 7 AND y = x WITH [
    'nextStart(p) := IF ( guard7(p,x) OR
                          ( NOT guard7(p,x) AND x'procNum(p) + 1 = N ) )
                     THEN rem(M)(x'nextStart(p)+Delta)
                     ELSE x'nextStart(p) ENDIF ,
    'procNum(p)   := IF ( guard7(p,x) OR
                          ( NOT guard7(p,x) AND x'procNum(p) + 1 = N ) )
                     THEN 0
                     ELSE x'procNum(p) + 1 ENDIF ,
    'tag(p)       := IF ( guard7(p,x) OR
                          ( NOT guard7(p,x) AND x'procNum(p) + 1 < N ) )
                     THEN rem(M)(x'tag(p)+1)
                     ELSE x'nextStart(p) ENDIF,
    'endSafe(p)   := IF ( NOT guard7(p,x) AND x'procNum(p) + 1 = N )
                     THEN rem(M)(x'nextStart(p)+Delta)
                     ELSE x'endSafe(p) ENDIF,
    'startSafe(p) := IF ( NOT guard7(p,x) AND x'procNum(p) + 1 = N )
                     THEN x'nextStart(p)
                     ELSE x'startSafe(p) ENDIF,
    'jump(p)      := IF guard7(p,x)
                     THEN x'jump(p) + 1
                     ELSE IF x'procNum(p) + 1 = N
                          THEN 0
                          ELSE x'jump(p) ENDIF
                     ENDIF,
    'pc(p) := 50
  ]
```

(Note that we also abbreviated the guard in line 7 in PVS to `guard7(p,x)`)

5.2 Proving invariants

Now that we have a framework to work with, we can move on to the actual proving part. Let us first look at invariant (14.), for which the proof is relatively short and easy. This allows us to show the basic principle behind the proof of each invariant with an easy example. Recall the definition of (14.):

(14.) `loc[X] = top`

Translating this to PVS is easy enough. We construct a function, `inv14(x): bool`, that takes a state x as argument and returns a boolean value that expresses whether x satisfies the conditions of predicate (14.) or not. Since in PVS we can refer to the variables in the state record x by using back quotes ', we can define the function as:

```
inv14(x): bool = x'loc( x'xshared ) = x'top
```

In order to prove that this condition holds under any changes made by the program, we have to test it against any changes in the state space x that is made under the different steps. This is where the step function comes in. Look at the following statement:

```
inv14_kept_valid: LEMMA
   inv14(x) AND step(p,x,y) IMPLIES inv14(y)
```

This is the declaration of the lemma we want to prove: if inv14_kept_valid can be proved correct we know that the expression inv14 stays true under any circumstances and thus is invariant. The lemma inv14_kept_valid can be read as 'if in the initial state x condition inv14 holds, and any process p changes the state from x to y, then in the resulting state y condition inv14 also holds'. The first part of this statement is important too and must not be forgotten: the invariant should hold initially. This means we also have to show the initial validity of the lemma. We discuss this in section 5.3.

The proof of the lemma inv14_kept_valid itself is hardly any work because it is such a trivial invariant. We can use the PVS command called (grind), a prover strategy that tries to automatically complete a proof branch by applying as much simplifications as it can. Using (grind) indeed verifies that the invariant is not invalidated by any of the steps.

It gets more interesting if the proofs become more complicated. Take the proof for invariant (23.) for example.

(23.) $start.r = \mathtt{loc}[q.r, k.r]$

The proof definition and lemmas formulated to complete the proof can be seen in figure 17. In the definition, the q and k have been replaced by their direct PVS counterparts[4].

It is not possible to prove the correctness of this formula by formulating a lemma like inv14_kept_valid consisting of only the definition of invariant (23.) and the step- function and simply telling PVS to (grind) this. If we try to do so, PVS leaves us with two subgoals that it can not prove without additional information. The first subgoal concerns line 1, and the second subgoal concerns line 8. Something must be happening there that threatens the validity of the invariant. Indeed, in line 1 $start.r, q.r$ and $k.r$ are set; and in line 8, a new loc is written. These issues that PVS raises have sometimes been thought of beforehand but it can also be the case that they were overlooked at first. In this, PVS is very useful to detect any flaws or omissions in a proof. For the proof of (23.), as stated in the textual description of the proof already, we should have enough information to prove the validity through lines 1 and 8 by using the information from invariants (14.) and (08.), respectively.

Since some parts of the proof require more work than the other, we split it into four lemmas instead of working with a single lemma. This is not a requirement for PVS – we could have chosen to construct a proof directly from a single lemma, and handle the more difficult proof branches in the subtree – but splitting it into logical chunks has several advantages. First of all, this makes it easier to maintain and edit the proof. A second advantage of this strategy is that you do not have to rerun parts of the proof that have already been proven, and that you are not interested in when returning to an unfinished proof. It also reduces the number of formulas in the sequent, making it easier to handle for both PVS and the human user.

[4] A note on the syntax of this definition: it would have been possible to use the (correct) definition x`loc(x`tag(r)) = x`start(r), and in fact we have done so at first. However, in the PVS version we used this led to problems later. Also see section 5.4

The first lemma is the 'list'-part, which should prove all trivial steps. Next are the lemmas for the steps that require extra information to prove. The last lemma is an 'overall' proof that combines the three first steps into one. We will explain the proofs step by step.

```
inv23(r,x): bool =
   x`loc(x`tag(r)`1, x`tag(r)`2) = x`start(r)
inv23_list: LEMMA
   inv23(r,x) AND step(p,x,y) IMPLIES inv23(r,y) OR
      step1(p,x,y) OR step8(p,x,y)
inv23_at1: LEMMA
   inv23(r,x) AND step1(p,x,y) AND inv14(x) IMPLIES inv23(r,y)
inv23_at8: LEMMA
   inv23(r,x) AND step8(p,x,y) AND inv08(r,x) IMPLIES inv23(r,y)
inv23_kept_valid: LEMMA
   inv23(r,x) AND inv14(x) AND inv08(r,x) AND step(p,x,y)
   IMPLIES inv23(r,y)
```

Fig. 17. Lemmas required to prove the correctness of invariant (23.)

`inv23_list`: The complicated steps have been appended to the consequent in the definition of the lemma. The 'list' part can then be proved by the `(grind)` command, but it's often worth it to postpone that a bit. A `(grind)` action is powerful but stupid, and can lead to a lot of 'useless' calculations, unnecessarily increasing CPU time.

The strategy we use for the `list`-parts depends on the amount of steps required by PVS to complete the proof, but it is most often done in the way depicted in figure 18. After simplifying the proof and hiding formulas 2 and 3 – this are the `step1`

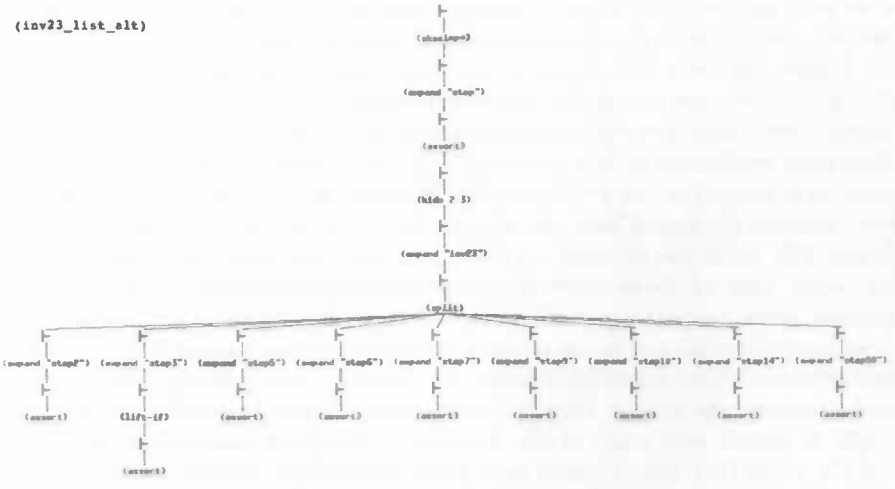

Fig. 18. Proof tree for `inv23_list`.

and `step8` in the implication – we split the proof by the possible `step` disjunctions.

The simplification from the beginning already removed steps 1 and 8 and only the trivial steps remain. They are shown to be true by expanding the definition of the particular steps and asserting their validity.

`inv23_at1` In the definition of this lemma we added invariant (14.) as a precondition of the step because it contains information we require to prove the validity of (23.) through line 1. Here as well we could choose to perform a `(grind)`job directly, but it is more efficient to do (some of) the work ourselves first. When enough information is present, `(grind)` does a good job most of the time if CPU usage is not a concern. However, it does not *always* work: on certain occasions the necessary logic step(s) are beyond the range of PVS decision procedures, and there can also be too many formulas with too many instantiatable variables leading to wrong instantiations.

The proof tree for `inv23_at1` is shown on the left side of figure 19. The proof is split in two cases, one in which $p = r$, and one in which $p \neq r$. If $p \neq r$, line 1 is not an issue and can quickly be asserted. The `(hide)` command given there is not required, but cleans up clutter irrelevant to the proof of the subgoal. In the first case with $p = r$, we substitute r for p using a replace command, and expand all definitions. This eventually leads to the following sequent:[5]

-1: (x)loc$[q, k]$ = (x)$start.r$
-2: $pc.r = 1$ AND
(y) := (x)**with**
$start.r$ = (x)top
$tag.r$ = (x)X
$ll.r$ = (x)loc[X]
$pc.r = 2$
-3: (x)loc[X] = (x)top

1: (y)loc[(y)$tag.r$] = (y)$start.r$

It is clear that the resulting state y satisfies the consequent if it follows the transformation from the -2 formula of the antecedent, because of the extra information the -3 formula brings. This assertion can be done by a simple `(assert)` command in PVS and will prove the subgoal for the formula.

`inv23_at8` The proof of validity in line 8 has been constructed a bit differently. The proof tree can be seen in figure 19 as well. It is split over the if-statement creating two branches in the proof, one in which the guard holds and one in which it doesn't hold. The easiest part is the right branch, representing the case in which the guard in line 8 fails. No relevant variables are changed and this branch can be easily `(assert)`-ed. The left branch, where the guard evaluates to true, is easy to `(assert)` as well once the information from invariant (08.) is made explicit.

`inv23_kept_valid` The `kept_valid` proofs combine the sub-steps that the other proofs provide. Figure 20 displays the proof tree. It consists of applying all of the other 'subproofs' to prove the whole. It should need no further explanation.

Proofs by implication Some predicates can be derived from other predicates that have already been proven. It may be possible to show their correctness just like we proved the other invariants, but doing so would probably mean repeating proof

[5] This notation has been copied from PVS and is a common notation in temporal logic reasoning.

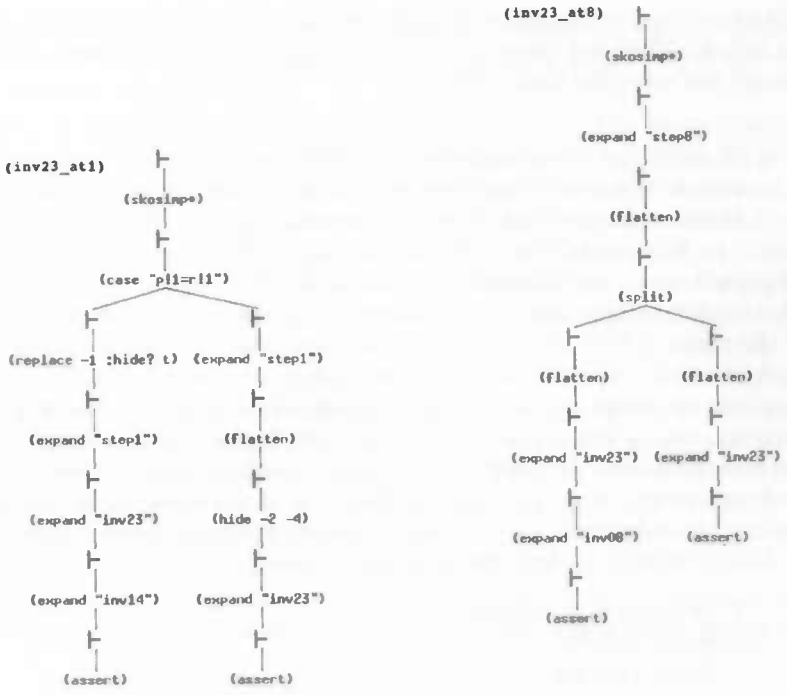

Fig. 19. Proof trees for `inv23_at1` and `inv23_at8`.

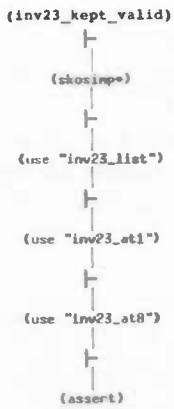

Fig. 20. Proof tree for `inv23_kept_valid`.

work that was already done earlier. It is easier to create proofs by implication for them. An easy example would be invariant (07.) that follows directly from (07a.). In PVS, we formulated this as:

```
inv07_implied: LEMMA
   inv07a(r,x) IMPLIES inv07(r,x)
```

We already proved (07a.)'s correctness, and by this implication we show that for every state x where (07a.) is true, (07.) is also true.

5.3 Loose ends

The whole proof has been translated into PVS this way and proved correct. Aside from the proof of all individual invariants, we have added two additional proof steps. The first is the initialization step, proving initial invariance of all invariants. The second is a 'completeness'-step, proving all invariants at once in one combined invariant.

To do this we need an combined expression of all invariants first. We define it as follows, with the dots representing all unnamed invariants in between:

```
all_invariants(x): bool = FORALL r:
   inv01(x) AND inv02(r,x) AND...
   ...AND inv35(r,x) AND inv36(r,x)
```

The initialization step requires little additional explanation. It is enough to introduce a new variable of the `state`-type, for which all the record entries (the 'actual' variables) have been given the initial values from figure 6. After that it is only a matter of proving the lemma:

```
all_invariants_init: LEMMA all_invariants(init)
```

The 'completeness'-step can be considered as a summarizing step in which we prove all invariants at once.

```
all_invariants_kv: LEMMA
   all_invariants(x) AND step(p,x,y) IMPLIES all_invariants(y)
```

It shows that all preconditions of all `kept_valid`-lemmas follow from the `all_invariants`-lemma. In the proof of `all_invariants_kv`, instead of doing all subproofs in detail again, we can use the `kept_valid` and `_implied` proofs we constructed earlier. This means that if we missed a definition or forgot a proof, we would also see that here.

For completeness' sake we should mention that there is a part of the correctness proof that we skipped. The proof would be finished if we prove by induction that an execution, starting in state `init` and repeatedly making steps, would not invalidate the invariant `all_invariants`. We omitted this step since it is fairly trivial once `all_invariants_kv` was proved.

5.4 Experiences with PVS

PVS is a powerful tool but like all programs it also has its limits. The prover is supplied with a standard library called the prelude. The prelude consists of definitions of theories, lemmas and data types that can be used in proofs.

Sometimes it is easy to overlook the fact that something that you take for obvious can not be solved by PVS because there are no rules present to handle that particular case. Even if they are present in the prelude, they still need to be called and used. For instance, in our proof this happened in cases where the lemmas used a mod-operation (invariants (03.), (15.), (16.), (29.) and (32.)).

There are already rules present in the library about modulo arithmetic but they are defined very generically. In this case we found it easier to construct our own lemma since we needed only 2 specific cases, one to prove that the a **mod** $2 \neq (a+1)$ **mod** 2 and one to prove that $(a+2)$ **mod** $2 = a$ **mod** 2.

Another problem has been a PVS bug in the substitution of a pair. This became apparent after various crashes of the PVS environment while trying to prove invariant (23.) that expresses a relation between $start.r$ and $\mathtt{loc}[q.r, k.r]$ or, $\mathtt{loc}[tag.r]$.

The ghost variable `loc` was defined as an array[processtype, seqnumtype] of seqnumtype. The program variable *tag* as a tagtype which was a record/pair of processtype and seqnumtype.

At first we had defined the invariant with `loc`[*tag.r*]. This led to a certain point in which PVS had to compare the `loc`-register from the invariant to the situation of the `loc`-register in a changed state. More specifically, it had to compare `loc`[*tag.r*] from state `x` to the same `loc` from state `y`, when in the meantime `loc`[p, *seq*.p] had changed, and failed at it by somehow failing a type check for this.

The PVS version we used was version 3.2, but this bug may have been fixed in a later PVS version (at the time of writing, the most recent version is PVS 4.0). We did not verify this.

A third problem with PVS was minor but subtle, and can be annoying at times. While PVS is running and attempting to prove invariants or goals, it keeps an internal record of things it has tried, attempted etc. These results are cached, and carry over between sessions as well. In some cases this can result in PVS failing to prove something it actually should be able to do, due to it making wrong assumptions based on previous attempts in the decision procedure.

It can also sometimes lead to a situation in which PVS thinks there are TCC's (Type Correctness Conditions) left to prove while this is actually not the case.

6 Conclusion

Using an assertional proof technique we have managed to achieve a small improvement in the algorithm with unbounded sequence numbers. The relevance of this improvement is limited since the algorithm is only really correct for *true* unbounded sequence numbers, something that can not be achieved.

It would have been desirable if the formal proof of the second algorithm had been completed. This 'proved' to be too much work. This reflects the usual problem for program correctness: a proof of correctness is often only attempted once the algorithm has already been completed, instead of during development, and proving correctness afterwards tends to be complex and impractical. The amount of work required to complete a proof seems to be proportional to the complexity of the problem, and this problem was rather complex.

Nevertheless, working on a detailed proof for such a long while has given us a better insight in the delicate workings of the algorithm. It has also given us the impression that the algorithm is indeed correct (but then again, *thinking* that something is correct does not necessarily mean that it *is*). One thing it at least taught us is that it is never a good idea to jump to conclusions: one moment you think you understand how it works, but the next moment it turns out to be different.

Given more time, we would also have wanted to improve the proof of the first algorithm. For example, by trying to eliminate (EXT.), and renaming the different lemmas leading to the correctness proof.

For possible future work, first of all the proof of the bounded algorithm could be completed. As we said above, it is our impression is that the algorithm is correct, and except for the three expressions on page 35 ((W1), (W9), and (W4short)) the proof has been completed. Next, it would be an idea to verify the correctness of the bounded algorithm using the "efficient selection algorithm" that was offered in [11]. Most likely, parts of the proof of the "simple" bounded algorithm could be reused for that, but it would also require a lot of new work, especially since the efficient selection function seems to require three atomic steps to complete.

References

1. M. Abadi and L. Lamport, The existence of refinement mappings, Theoretical Computer Science 82 253—284, 1991
2. J. H. Anderson and M. Moir, Universal Constructions for Multi-Object Operations, Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, p.184–194, August 1995
3. M. Clint, Program Proving: Coroutines, Acta Informatica 2, p.50—63, 1973
4. E. W. Dijkstra, A Discipline of Programming. Prentice-Hall Series in Automatic Computation, 1976, ISBN 0-13-215871-X
5. E. W. Dijkstra, 1968b. A constructive approach to the problem of program correctness, BIT 8 (1968), 174—186.
6. E. W. Dijkstra, Cooperating sequential processes, in F. Genuys (ed.), Programming Languages, Academic Press, New York, 1968.
7. M. Herlihy, A methodology for implementing highly concurrent data objects. ACM Transactions on Programming Languages and Systems, 15(5):745—770, 1993.
8. Wim H. Hesselink and Jan Eppo Jonker, A Refinement Proof of a Multiword LL/SC Object. Report and PVS files at `http://www.cs.rug.nl/~wim/pub/mans.html`, 2006.
9. C. A. R. Hoare, An axiomatic basis for computer programming. Communications of the ACM, 12(10):576—585, October 1969.
10. Ting-Lu Huang, Jann-Hann Lin, An assertional proof of a lock synchronization algorithm using fetch and store atomic instructions. International Conference on Parallel and Distributed Systems, p.759—768, 19-22 Dec 1994
11. P. Jayanti, S. Petrovic, Efficient and Practical Constructions of LL/SC Variables. Proceedings of the twenty-second annual symposium on Principles of distributed computing, p.285—294, July 13-16, 2003, Boston, Massachusetts
12. Leslie Lamport, The 'Hoare Logic' of Concurrent Programs. Acta Informatica 14, 21—37 (1980)
13. Leslie Lamport, "The writings of Leslie Lamport" `http://research.microsoft.com/users/lamport/pubs/pubs.html`, various dates.
14. S. Owicki, D. Gries, An Axiomatic Proof Technique for Parallel Programs I. Acta Informatica 6, 319—340, 1976.
15. PVS System Guide Version 2.4, `http://pvs.csl.sri.com/documentation.shtml`