A Refinement Proof of a Multiword LL/SC Object

Wim H. Hesselink and Jan Eppo Jonker, 24 April 2006
Dept. of Mathematics and Computing Science, University of Groningen
P.O.Box 800, 9700 AV Groningen, The Netherlands
Email: w.h.hesselink@rug.nl, j.e.jonker@rug.nl
Web: http://www.cs.rug.nl/~wim

Abstract. The algorithm of Jayanti and Petrovic (ICDCS 2005) gives a wait-free implementation of load-linked/store-conditional (LL/SC) for multiword variables, given LL/SC actions on single words. The authors gave a behavioural proof of correctness. We present a refinement proof that has been verified with the proof assistant PVS. We give an improved algorithm which needs fewer single-word LL/SC registers and fewer shared multiword variables, and in which one single-word LL operation has been eliminated. We also present a pure version in which all accesses to the LL/SC variables are done by LL, SC, or VL.

Keywords: load-linked/store-conditional, refinement, forward simulation, PVS, history variable, safe variable.

1 Introduction

In [JP05], Jayanti and Petrovic develop an efficient wait-free implementation of LL/SC of a multiword, given LL/SC for single word variables. Load-Linked and Store-Conditional (LL/SC) form a pair of instructions that is very useful for the implementation of lock-free (or non-blocking) concurrent algorithms.

The LL operation for an object $\mathcal{O}$ reads the value of $\mathcal{O}$ and gives p "a link to object $\mathcal{O}$". When a process p calls the SC operation of $\mathcal{O}$ while it has a link to $\mathcal{O}$, it assigns a new value to $\mathcal{O}$, removes all links to $\mathcal{O}$, and returns *true*. When p calls SC without a link to $\mathcal{O}$, the call does nothing but returns *false*. A boolean function VL indicates that process p has a link to $\mathcal{O}$. We refer to [JP05] for a discussion of the relevance of LL/SC and the availability of it on modern machines.

Recall that a shared variable for concurrent processes is called *safe* [Lam86] if every read operation that does not overlap with any write operation returns the most recently written value and every write operation that does not overlap with any other write operation successfully writes; in all remaining cases of concurrent reading or writing, the value written or delivered must always be a legitimate value of the domain of the variable. The primary example of a safe variable is a multiword, i.e., an array the elements of which can be read or written atomically, while interference may occur during reading or writing of the entire array.

Given LL/SC/VL for single machine words, the paper [JP05] gives a wait-free implementation of LL/SC/VL for safe variables of arbitrary size, together with a proof of correctness.

The proof in [JP05] is a behavioural proof, i.e., based on the analysis of execution sequences. It appears to be a complete proof, but we were not able, let alone eager, to follow it, since it requires the reader to consider the values of several variables during complicated time intervals. We therefore devised a refinement proof of the algorithm, primarily based on invariants. During the design of the invariants, we heavily relied on the proof assistant PVS [OSRSC01], to keep track of the proof obligations, and not to be fooled by wishful thinking.

During this investigation, but after we had completed the first version of the proof, we obtained two simplifications of the algorithm. Firstly, we eliminated one

of the implementing LL operations. Secondly, if the word length is big enough, we reduced the space complexity of the algorithm by a factor 2/3. Fortunately, after minor modification, our proof also proved the simplification. In this note, we thus present the simplified algorithm with a refinement proof that has been verified with PVS.

If N is the number of processes, our version of the algorithm needs a word length K with $N(N+1)(2N+1) \leq 2^K$ and it requires $2N+2$ single-word LL/SC registers and a shared array of $2N+1$ safe variables. The version of [JP05] needs $6N^2 \leq 2^K$ and requires $3N+1$ single-word LL/SC registers and a shared array of $3N$ safe variables. We present the proof in such a way that it applies to both versions of the algorithm.

The practical relevance of the algorithm and its improvement is limited. The fact that it is wait-free rather than lock-free is nice, but must not be over-estimated since the wait-freedom is lost in most applications of the higher-level LL/SC operations. If one needs more than one LL/SC variable of the higher level, one must realize that there are cheaper and more powerful implementations of arrays of lock-free (non-blocking) multiword variables [DHLM04,GH04]. For example, [GH04] gives a lock-free implementation of atomic actions on K multiword variables by means of $N+K$ safe variables and K single word LL/SC variables.

Nevertheless, the algorithm and its proof show a kind of buffer management that is interesting in itself and may have important other applications.

Overview. In Section 2 we present our improved version of the algorithm of [JP05]. In Section 3 we give two specifications of LL/SC, one to be used for the single-word version, and the other for the multiword version. In Section 4, we reformulate the code for the ease of formal manipulation. In Section 5, we provide an overview of the proof and a description of the role of the proof assistant PVS in it.

In Section 6, we establish the main uniqueness invariants needed to preclude interference. In Section 7, we derive invariants about the values transferred and the delay of the processes. In Section 8, we determine the synchronization point of the multiword LL operation, and prove the related invariants. Section 9 contains the forward simulation from the reformulated algorithm to the second specification of LL/SC.

In Section 10, we treat a pure variation of the algorithm in which all atomic accesses are performed with LL, SC, and VL. In Section 11, we draw our mixed conclusions.

2 The Revised Algorithm of Jayanti and Petrovic

We first present our version of the algorithm in such way that it can easily be compared with [JP05]. We use the convention that shared variables are in typewriter font and that private variables are slanted. Outside of the code of process p, we write $v.p$ to denote the value of the private variable v of p. We use type *Item* for the type of the safe variables or multiwords, and give the processes persistent private variables

$$val, arg : Item \ ,$$
$$result : Bool \ .$$

The value read by *LL* will be stored in *val*. The variable *arg* holds the value to be stored for *SC*. The boolean result of *SC* and *VL* will be kept in *result*.

Recall that N is the number of processes. Let M be any number with $M > N$. The cheapest choice is $M = N+1$, but by allowing $M > N$ we can get a simultaneous proof of the original version of [JP05]. The algorithm uses the following three ranges:

$$Process = 0 \ldots N-1 \ ,$$

$$Number = 0 \ldots M-1 \ ,$$
$$Index = 0 \ldots M+N-1 \ .$$

The algorithm uses the shared variables

$$\texttt{xx} : Index \times Number \times Process \ ,$$
$$\texttt{buf} : \textbf{array}\ Index\ \textbf{of}\ Item \ ,$$
$$\texttt{bank} : \textbf{array}\ Number\ \textbf{of}\ Index \ ,$$
$$\texttt{help} : \textbf{array}\ Process\ \textbf{of}\ Bit \times Index \ .$$

The algorithm is based on LL/SC of the single words `xx` and `bank`$[k]$ and `help`$[p]$. To explain the algorithm, we split `xx` into its components and introduce the shared variables

$$\texttt{xind} : Index \ , \ \texttt{xnr} : Number \ , \ \texttt{xhpr} : Process \ ,$$

with the convention that `xind`, `xnr`, `xhpr` are always the three components of `xx`. The processes have the private persistent variables:

$$ind, mybuf : Index \ ,$$
$$nr : Number \ ,$$
$$hpr : Process \ .$$

The first idea is that `buf`[`xind`] always holds the current value of the object. The tuple (ind, nr, hpr) serves as a private copy of `xx`. The index *mybuf* is a private candidate for the new value of `xind` after a successful SC operation.

Since delayed processes may use obsolete values of `xind`, the value of `xind` is not discarded after usage but stored in `bank`[`xnr`] (in line 33 below). The `bank` thus serves to protect recent history so that delayed processes are served correctly, though possibly with outdated results.

Array `help` is used to ensure that only recent history needs to be protected. A storing process, say p, *helps* (in line 35 below) process $hpr.p$ if that process is delayed in reading, by providing it with a pointer to a correct but outdated item.

These three ideas are due to [JP05]. Our main contribution to the design is to undo the decision of [JP05] that $M = 2N$ and $hpr = nr$ **mod** N.

The initial values are `xx` $= (0,0,0)$, `buf`$[0]$ $=$ *item0*, the initial value of the multiword, `bank`$[k] = k$ for all numbers k and $mybuf.p = M + p$ and $ind.p = nr.p = hpr.p = fst($`help`$[p]) = 0$ for all processes p.

We now present the algorithm, with the names *LLmul*, *SCmul* and *VLmul* for the three implemented procedures on a multiword. We use *fst* and *snd* to get the first and second component of a pair. In comparison with [JP05], we shift the numbers of the locations by 10 or 20 to make them easier to recognize.

```
     proc LLmul(p)
11:      help[p] := (1, mybuf)
12:      (ind, nr, hpr) := LL(xx)
13:      val := buf[ind]
14:      (a, b) := LL(help[p]) ;  if a = 0 then
15:          (ind, nr, hpr) := LL(xx)
16:          val := buf[ind]
17:          if ¬VL(xx) then
18:              val := buf[b]  end
19:      else SC(help[p], (0, b))  end
20:      mybuf := snd(help[p])
21:      buf[mybuf] := (safe) val .
```

The lines 21 and 37 below contain nonatomic assignments to safe variables. This is indicated by the keyword **(safe)**. Reading from a safe variable, as in 13, 16, 18, can be regarded as atomic, but may yield an arbitrary value when there is an interleaved write action on this variable.

```
        proc SCmul(p)
32:         if LL(bank[nr]) ≠ ind ∧ VL(xx) then
33:             SC(bank[nr], ind) end
34:         (a, d) := LL(help[hpr]) ;   if a = 1 ∧ VL(xx) then
35:             if SC(help[hpr], (0, mybuf)) then
36:                 mybuf := d end end
37:         buf[mybuf] := (safe) arg
38:         e := bank[(nr + 1) mod M]
39:         if SC(xx, (mybuf, (nr + 1) mod M, (hpr + 1) mod N)) then
40:             mybuf := e
41:             result := true
42:         else result := false end .

        proc VLmul(p)
45:         result := VL(xx) .
```

Since the current value of the object is kept in `buf[xind]`, the store action of *SCmul*(p), if it occurs, is split over the lines 37 and 39. The decision to store depends on whether p still has a link to `xx`. Reading the current value also requires two actions. In *LLmul*(p), process p tries to read the current value in 12 and 13. In line 14, it tests whether in the mean time some other process q has executed line 35 with $hpr.q = p$. If so, p tries again to read the value in the lines 15 and 16. If the indirection read in 15 is obsolete when p evaluates the guard of 17, process p reads the value reserved for it by process q in 35. Indeed, if a process q executes line 35, it provides process $hpr.q$ a pointer to a value of the object that q has stored in line 21.

Since, in line 18, delayed readers may be reading `buf`$[b]$ for obsolete indices b, a number of obsolete indices are protected in array `bank`. This array is used as a circular buffer. Obsolete indices that need no protection anymore, are recycled in line 38.

In view of the test *VL*(`xx`) in 34, a process p helps another process only when executing *SCmul* with a link to `xx`. The process it helps has number $hpr.p$, a copy of `xhpr`. Since all processes deserve help, `xhpr` is incremented modulo N for each fresh value of `xx` in line 39. It follows that, if a process p remains long enough in the stretch from 12 to 14, some other process will help it and reset the first bit of `help`$[p]$. This implies that not more than $N + 1$ indices have to be protected in `bank`. It is therefore enough to take $M > N$. This concludes a rough description of the algorithm. The proof below of course provides many additional details.

We next discuss the grain of atomicity, which is indicated by the line labels. Apart from the assignments to safe variables, all labelled statements are atomic. In Section 4, we also model the assignments to safe variables by atomic commands.

For the ease of proving, it is advantageous to have the atomic commands as big as possible, since this makes the underlying transition system smaller. On the other hand, the straightforward implementation of a command that contains more than one reference to a simple shared variable is not atomic [OG76]. We therefore need to justify the atomicity of the commands 32 and 34, which refer to two shared variables (`xx` and either `bank`$[nr]$ or `help`$[hpr]$).

In both cases, it concerns the test of a conjunction. It is supposed that the lefthand conjunct is evaluated before the righthand conjunct. If either conjunct evaluates to false, the conjunction is false at that moment. If both conjuncts evaluate

to true, the conjunction was true when the lefthand conjunct was evaluated. Indeed, once $VL(\texttt{xx})$ is false, it remains false until the process itself calls $LL(\texttt{xx})$.

A next point of doubt in the above code could be the atomic assignment to the LL/SC variable $\texttt{help}[p]$ in line 11. If a machine architecture offers LL/SC/VL on some variables, it need not also offer atomic assignments to them. In such a case, line 11 can be replaced by

10: $LL(\texttt{help}[p])$;
11: $SC(\texttt{help}[p], (1, mybuf))$;

Here the assignment is split into two atomic commands and the results of *LL* and *SC* are ignored. Also, note that the LL/SC variables $\texttt{help}[p]$ and $\texttt{bank}[k]$ are read atomically in the lines 20 and 38. If necessary, this can be implemented with *LL*. In this way, one obtains a pure LL/SC algorithm in which the variables xx, $\texttt{help}[p]$, and $\texttt{bank}[k]$ are accessed only via LL, VL, or SC. The correctness of this pure alternative is shown below in Section 10.

The differences with [JP05] are as follows. Firstly, in [JP05], line 19 is preceded by a separate LL operation, but this is unnecessary. The more important difference is that [JP05] takes $M = 2N$ and does not have the variables xhpr and *hpr*. This is possible since our version then has the invariants $\texttt{xhpr} = \texttt{xnr}$ **mod** N and $hpr.q = nr.q$ **mod** N. This makes the word xx smaller, but makes array bank longer than when one takes $M = N + 1$.

3 Specification of LL/SC

In order to prove the correctness of the algorithm presented, we need a formal specification of LL/SC. We also need this for the occurrences of single-word LL/SC in the implementation. The simplest specification [GH04] of LL/SC for a given shared variable it of type T attaches to it a set of processes itSet, which contains the processes that have loaded the value of it since the latest successful *SC* operation. So, initially, $\texttt{it} = item0$ and itSet is empty. The three procedures are specified by

proc $LL(p)$
11: $val := \texttt{it}$; $\texttt{itSet} := \texttt{itSet} \cup \{p\}$

proc $SC(p)$
32: **if** $p \in \texttt{itSet}$ **then** $\texttt{it} := arg$; $\texttt{itSet} := \emptyset$; $result := true$
else $result := false$ **end**

proc $VL(p)$
45: $result := (p \in \texttt{itSet})$

Here, *val* and *arg* are private variables of process p of type T and *result* is a private boolean variable of p. As indicated by the labels, all three procedures are regarded as atomic. Let us call this the simple specification *LSV*.

For the proof of the algorithm of [JP05] we need a more elaborate specification *ELSV* that makes the history of the variable it explicit in a shared function $\texttt{hist} : \mathbb{N} \to T$ with a shared variable $\texttt{top} : \mathbb{N}$ such that hist(top) is the current value of it. Variable top is incremented by every successful *SC* operation. Private variables *start* and *ll* serve to record the value of top at the moment that *LL* is called and is executed, respectively.

proc $LLe(p)$
11: $start := \texttt{top}$
12: **choose** ll **with** $start \leq ll \leq \texttt{top}$; $val := \texttt{hist}(ll)$

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

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

In this case, *SCe* and *VLe* are atomic, while *LLe* consists of two atomic statements. Since *VL* must be false initially, we require that initially `top` > 0 and $ll.p = 0$ for all p. Moreover, `hist(top)` $=$ *item0*, the initial value of `it`.

It can be proved by extending the techniques of [Hes05] that the specifications *LSV* and *ELSV* are equivalent: each of them implements the other in the sense of [AL91]. They correspond via the equalities `it` $=$ `hist(top)` and `itSet` $= \{q \mid ll.q =$ `top`$\}$. The main problem is to deal with the non-atomicity of *LLe*. We have to leave this as an exercise for the theoretically minded reader.

4 Reformulation of the Code

We now reformulate the code of Section 2 to recognize in it specification *ELSV* of Section 3. Specification *LSV* of Section 3 is used to decode the low level LL/SC operations on `xx`, `bank`$[k]$ and `help`$[q]$. We introduce sets of processes `xSet`, `baSet`$[k]$ and `heSet`$[q]$ to play the roles of `itSet` for these variables, respectively.

With respect to the assignments to the safe variables in lines 21 and 37, there are two measure to take: we have to preclude write-write interferences and to model the read-write interferences. To preclude write-write interferences in 21 and 37, it suffices to prove the following invariant:

(Wq) $\quad pc.q > 20 \ \wedge \ pc.r > 20 \ \wedge \ mybuf.q = mybuf.r \quad \Rightarrow \quad q = r \ .$

This will be a consequence of the invariant (Iq0) to be proved in Section 6 below.

As for read-write interferences, recall that reading a safe shared variable while writing is in progress, may yield an arbitrary value of the correct type. We therefore model a labelled assignment $\ell :$ `x` $:=$ (**safe**)E as a nondeterministic choice

$$\ell : \quad (\texttt{x} := \text{arbitrary ; } \texttt{goto } \ell) \quad [\!] \quad \texttt{x} := E \ .$$

Here, the first alternative leads to looping. We therefore postulate weak fairness, so that the first alternative is not chosen infinitely often. This pattern is used to reformulate commands 21 and 37 in the code below.

The main problem is to identify the value *ll* that determines the synchronization point of *LL* such that the values read by *LLmul* and *LLe* correspond. This problem is postponed to Section 8. We first need to derive a list of invariants.

To formulate these invariants conveniently, we split `help`$[q]$ into the two components

$$\texttt{aux} : \textbf{array } \mathit{Process} \textbf{ of } \mathit{Bool} \ ,$$
$$\texttt{hind} : \textbf{array } \mathit{Process} \textbf{ of } \mathit{Index}$$

with the convention that `aux`$[q]$ and `hind`$[q]$ are always the first and second components of `help`$[q]$. Here we identify the types *Bit* and *Bool* in the usual way with $0 =$ *false* and $1 =$ *true*. We also use the components `xind`, `xnr`, `xhpr` of `xx`, as introduced in Section 2. We introduce one private variable *bb* to hold the numbers b, d, e read in the lines 14, 34, 38, respectively.

We introduce a location 50 from which the procedures *LLmul*, *SCmul*, and *VLmul* can be called, to model that the procedures can be called by the processes in arbitrary order.

calling(p)
50: (**goto** 11 ‖ **goto** 45 ‖ **choose** *arg* , **goto** 32) .

The variables *start*, `top`, and `hist` of specification *ELSV* are treated as auxiliary variables [OG76] or history variables [AL91]. The actions on them are inserted at appropriate places. We provide all processes with history variables *ptop* to hold the value of `top` at the moment that the lines 12 and 15 are executed. In this way the code of Section 2 is transformed into:

proc *LLmul*(p)
11: *start* := `top` ; `aux`[p] := *true* ; `hind`[p] := *mybuf* ;
12: *ind* := `xind` ; *nr* := `xnr` ; *hpr* := `xhpr` ; *add*(p, `xSet`) ; *ptop* := `top` ;
13: *val* := `buf`[*ind*] ;
14: *bb* := `hind`[p] ; *add*(p, `heSet`[p]) ; **if** `aux`[p] **then goto** 19 **end** ;
15: *ind* := `xind` ; *nr* := `xnr` ; *hpr* := `xhpr` ; *add*(p, `xSet`) ; *ptop* := `top` ;
16: *val* := `buf`[*ind*] ;
17: **if** $p \in$ `xSet` **then goto** 20 **end** ;
18: *val* := `buf`[*bb*] ; **goto** 20 ;
19: **if** $p \in$ `heSet`[p] **then**
`aux`[p] := *false* ; `hind`[p] := *bb* ; `heSet`[p] := $\emptyset$ **end** ;
20: *mybuf* := `hind`[p] ;
21: (`buf`[*mybuf*] := arbitrary ; `goto` 21) ‖ `buf`[*mybuf*] := *val* ; **goto** 50 .

It is well-known that actions on private variables can be atomically combined with actions on shared variables since there is no danger of interference. It is therefore allowed to combine line 36 with line 35, and the lines 40, 41 and 42 with 39. We do this to make the invariants easier to formulate and prove.

proc *SCmul*(p)
32: *add*(p, `baSet`[*nr*]) ; **if** `bank`[*nr*] = *ind* $\vee$ $p \notin$ *xSet* **then goto** 34 **end** ;
33: **if** $p \in$ `baSet`[*nr*] **then** `bank`[*nr*] := *ind* ; `baSet`[*nr*] := $\emptyset$ **end** ;
34: *bb* := `hind`[*hpr*] ; *add*(p, `heSet`[*hpr*]) ;
if $\neg$ `aux`[*hpr*] $\vee$ $p \notin$ `xSet` **then goto** 37 **end** ;
35: **if** $p \in$ `heSet`[*hpr*] **then**
`aux`[*hpr*] := *false* ; `hind`[*hpr*] := *mybuf* ;
`heSet`[*hpr*] := $\emptyset$; *mybuf* := *bb* **end** ;
37: (`buf`[*mybuf*] := arbitrary ; `goto` 37) ‖ `buf`[*mybuf*] := *arg* ;
38: *bb* := `bank`[(*nr* + 1) **mod** *M*] ;
39: **if** $p \in$ `xSet` **then**
`xind` := *mybuf* ; `xnr` := (*nr* + 1) **mod** *M* ;
`xhpr` := (*hpr* + 1) **mod** *N* ; `xSet` := $\emptyset$; *mybuf* := *bb* ;
result := *true* ; `top++` ; `hist`(`top`) := *arg* ;
else *result* := *false* **end** ; **goto** 50 .

Since *VLmul* is implemented as *VL*(`xx`), the action *SCmul* is successful if and only if *SC*(`xx`) in line 39 is successful. We therefore place the auxiliary modifications of `hist` and `top` in line 39.

proc *VLmul*(p)
45: *result* := ($p \in$ `xSet`) ; **goto** 50 .

We regard it as self-evident that the algorithm of Section 2 implements the transformed algorithm of this section.

5 Proving the Algorithm

When we started to investigate the algorithm, it soon became apparent that we could use PVS as a proof assistant. In PVS, we defined the state space in terms of the shared and private variables, like the following:

```
N, M: posnat
Process: TYPE = below[N]
Index: TYPE = below[N+M]

state: TYPE = [#
    xind: Index , % more shared variables
    ind: [Process -> Index] , % more private variables
    pc: [Process -> nat]
#]
```

The code of Section 4 is easily transformed into a transition system. For example, using `x` and `y` of type `state` and `p` of type `Process`, line 20 is represented by the definition:

```
step20(p, x, y): bool =
    x`pc(p) = 20 AND
    y = x WITH [
        `mybuf(p) := x`hind(p) ,
        `pc(p) := 21 ]
```

We then started to guess and prove several invariants as described in the next sections. This improved our understanding and our confidence in the correctness of the algorithm. The main creative step was to invent specification *ELSV* of Section 3, and its incorporation in the code of Section 4, especially in the lines 11 and 39.

Finding invariants in an algorithm one does not really understand requires a good intuition, but is mainly a lot of work. It was difficult to get the precise forms of the uniqueness invariants (Iq0) up to (Iq4) of Section 6. A next bottleneck was the set of invariants about delayed processes, starting with (Kq4) in Section 7.

The most difficult point was the identification of the synchronization point at line 34, and the associated introduction of the auxiliary variables `htop` and *syn*, as described in Section 8 below. It came as a surprize that we needed a forward simulation rather than a refinement mapping to conclude the proof in Section 9.

The complete PVS code is available at [Hes06]. Here, one can also see which invariants are used, and where, to prove preservation of some invariant. For example, the following lemma shows that (Jq10) is used to prove preservation of (Mq2) when process p executes line 18:

```
mq2_at_18: LEMMA
    mq2(q, x) AND jq10(q, x) AND step18(p, x, y)
    IMPLIES mq2(q, y)
```

The role of PVS was plain verification. We ourselves invented the invariants and the forward simulation. In the more difficult proofs of preservation of some invariants, we also had to guide the choices of case distinctions.

Overview of the proof. We prove that the transformed algorithm of Section 4 implements specification *ELSV* of Section 3 in the following way. In Section 6, we develop a list of invariants that preclude interference. In Section 7, the list is extended with invariants about values and delay. Section 8 determines the choice of *ll* in *LLe* and again extends the list of invariants. Finally, in Section 9, we construct a forward simulation from the algorithm as extended in Section 8 to specification *ELSV*.

6 The Uniqueness Invariants

The paper [JP05] mainly uses behavioural arguments, but it does have two invariants. The first one indicates how the $M + N$ index values can be distributed over the variables `xind`, *mybuf*.q, `hind`[q], and `bank`[k]. In order to obtain such a result, we started to prove a number of invariants about the controle variables that would enable us to prove something like this invariant.

In the invariants we implicitly universally quantify over all free variables like process identifiers q, r. The first main invariant is

(Jq0) $\quad q \in \texttt{xSet} \;\Rightarrow\; ind.q = \texttt{xind} \;\wedge\; nr.q = \texttt{xnr} \;\wedge\; hpr.q = \texttt{xhpr}$.

This holds initially, since then `xSet` is empty. The only locations where `xSet` and the other six variables are modified are 12, 15, and 39. Predicate (Jq0) is preserved when process q executes 12 or 15 since the consequent is made true. It is preserved at 39 since `xSet` is made empty. We use names like (Jq0) for the invariants to make them easier to find and replace during the development of the mechanical proof.

Two easy invariants concerning `aux` are

(Jq1) $\quad \texttt{aux}[q] \;\Rightarrow\; pc.q \in \{12, 13, 14, 19\}$,
(Jq2) $\quad pc.q = 19 \;\wedge\; \texttt{aux}[q] \;\Rightarrow\; q \in \texttt{heSet}[q]$.

Indeed, we use (Jq2) to prove that (Jq1) is preserved in 19. The invariance of (Jq2) uses the observation that when some process p at 35 makes `heSet`[q] empty it also resets `aux`[q].

Command 19 is needed to reset `aux`[p]. This is harmless for `hind`[p] because of the easy invariant

(Jq3) $\quad pc.q = 19 \;\wedge\; q \in \texttt{heSet}[q] \;\Rightarrow\; bb.q = \texttt{hind}[q]$.

We next turn to the actions of *SCmul*. The assignment of 33 uses correct private copies of `xnr` and `xind` according to

(Jq4) $\quad pc.q = 33 \;\wedge\; q \in \texttt{baSet}[nr.q]$
$\Rightarrow\; ind.q = \texttt{xind} \;\wedge\; nr.q = \texttt{xnr} \;\wedge\; hpr.q = \texttt{xhpr} \;\wedge\; \texttt{bank}[\texttt{xnr}] \neq \texttt{xind}$.

In order to show the invariance of (Jq4), we also postulate

(Jq5) $\quad pc.q \in \{34 \ldots 39\} \;\wedge\; q \in \texttt{xSet} \;\Rightarrow\; \texttt{bank}[\texttt{xnr}] = \texttt{xind}$,
(Jq6) $\quad pc.q = 33 \;\wedge\; q \in \texttt{xSet} \;\wedge\; q \notin \texttt{baSet}[nr.q] \;\Rightarrow\; \texttt{bank}[\texttt{xnr}] = \texttt{xind}$.

Preservation of (Jq4) follows from (Jq0) and (Jq5), preservation of (Jq5) follows from (Jq0), (Jq4), and (Jq6); preservation of (Jq6) follows from (Jq0) and (Jq4).

The assignments in 35 reset `aux`[r] and interchange *mybuf*.p and `hind`[$hpr.p$] because of the invariant

(Jq7) $\quad pc.q = 35 \;\wedge\; q \in \texttt{heSet}[hpr.q]$
$\Rightarrow\; \texttt{aux}[hpr.q] \;\wedge\; bb.q = \texttt{hind}[hpr.q]$.

Preservation of (Jq7) at 11 follows from (Jq1).

In 39, the new value of *mybuf* is taken out of `bank` according to

(Jq8) $\quad pc.q = 39 \;\wedge\; q \in \texttt{xSet} \;\Rightarrow\; bb.q = \texttt{bank}[(\texttt{xnr} + 1) \;\mathbf{mod}\; M]$.

Preservation of (Jq8) follows from (Jq0) and (Jq4).

We are now ready for invariant (I1) in Section 3 of [JP05]. We treat it as a conjunction of five invariants, to be called the uniqueness invariants. In order to express them succinctly, we use a conditional expression to define

$$\texttt{mb}.q = (pc.q \in \{12 \ldots 20\} \,?\, \texttt{hind}[q] : mybuf.q) \ .$$

Roughly speaking, the uniqueness invariants say that `xind`, the N indices $\texttt{mb}.q$, and the M indices $\texttt{bank}[j]$ are all different. Since there are only $M + N$ indices, this cannot be literally true. There is and must be one potential exception. It turns out that this is at $j = \texttt{xnr}$. The uniqueness invariants are

(Iq0) $\texttt{mb}.q = \texttt{mb}.r \quad \Rightarrow \quad q = r$,
(Iq1) $\texttt{mb}.q = \texttt{bank}[j] \quad \Rightarrow \quad j = \texttt{xnr}$,
(Iq2) $\texttt{bank}[j] = \texttt{xind} \quad \Rightarrow \quad j = \texttt{xnr}$,
(Iq3) $\texttt{bank}[j] = \texttt{bank}[k] \quad \Rightarrow \quad j = k$,
(Iq4) $\texttt{mb}.q \neq \texttt{xind}$.

First notice that (Iq0) implies predicate (Wq) of Section 4.

These five invariants are threatened by the assignments to `hind`, *mybuf*, `bank`, `xind`, and `xnr` in 19, 33, 35, and 39. The lines 11 and 20 are harmless because of the definition of `mb`.

In line 19, the assignment to $\texttt{hind}[p]$ (i.e., $\texttt{mb}.p$) is harmless because of (Jq3). In line 33, the assignment to $\texttt{bank}[nr.p]$ threatens (Iq1), (Iq2), and (Iq3). By (Jq4), we then have $nr.p = \texttt{xnr}$ and $ind.p = \texttt{xind}$. This is enough to preserve (Iq1) and (Iq2); in the case of (Iq3), we additionally need (Iq2) in the precondition.

In line 35, the invariants (Iq0), (Iq1), (Iq4) are threatened by the assignments to $mybuf.p$ and $\texttt{hind}[hpr.p]$. Here, we need (Jq7) and (Jq1) to ascertain that command 35 effectively swaps $\texttt{mb}.p$ and $\texttt{mb}.(hpr.p)$.

In line 39, the invariants (Iq0), (Iq1), (Iq2), (Iq4) are threatened by the assignments to $mybuf.p$, `xind`, and `xnr`. By (Jq0), (Jq5), and (Jq8), the effect of the body of 39 on the invariants (Iq*) is equivalent with the rotation

$$\{\ \texttt{xind} = \texttt{bank}[\texttt{xnr}]\ \} \quad \texttt{xind} := \texttt{mb}.p\ ;$$
$$\texttt{xnr} := (\texttt{xnr} + 1)\ \textbf{mod}\ M\ ; \quad \texttt{mb}.p := \texttt{bank}[\texttt{xnr}]\ .$$

Therefore (Iq0) is preserved because of (Iq1); (Iq1) is preserved because of (Iq3) and (Iq4); (Iq2) is preserved because of (Iq1) and (Iq4); (Iq4) is preserved because of (Iq0) and (Iq1).

In the next section, we also need the invariants:

(Jq9) $pc.q \in \{15, 16, 17, 18\} \quad \Rightarrow \quad bb.q = \texttt{hind}[q]$,
(Jq10) $pc.q = 18 \quad \Rightarrow \quad q \notin \texttt{xSet}$.

Preservation of (Jq9) follows from (Jq1) and (Jq7) at 35; (Jq10) is easy.

7 Invariants about Values and Delay

We come to the layer of invariants that express how the multiwords are represented. They are made recognizable by names that start with "Mq". Firstly, the latest value written arrives in `buf` at the main index `xind`, according to the invariants:

(Mq0) $\texttt{buf}[\texttt{xind}] = \texttt{hist}(\texttt{top})$,
(Mq1) $pc.q \in \{38, 39\} \quad \Rightarrow \quad \texttt{buf}[mybuf.q] = arg.q$.

Preservation of (Mq0) follows from (Iq4) at 21 and 37, and at 39 from (Mq1). Preservation of (Mq1) follows from (Iq0).

When its latest LL has not yet been overwritten, process q often holds the most recently written value in $val.q$ or $\texttt{buf}[mybuf.q]$, as is expressed in

(Mq2) $q \in \texttt{xSet} \ \wedge \ pc.q \in \{14, 17, 19, 20, 21\} \quad \Rightarrow \quad val.q = \texttt{hist}(\texttt{top})$,
(Mq3) $q \in \texttt{xSet} \ \wedge \ pc.q \in \{45, 50, 32, 33, 34, 35\} \quad \Rightarrow \quad \texttt{buf}[mybuf.q] = \texttt{hist}(\texttt{top})$.

Preservation of (Mq2) at 13 and 16 follows from (Jq0) and (Mq0), at 18 from (Jq10). Preservation of (Mq3) at 21 and 37 follows from (Iq0) and (Mq2).

Recall from Section 3 that we need to initialize `top` > 0. For simplicity, we choose `top` $= M \cdot N$ initially. It follows that `xnr` and $nr.q$ represent `top` and $ptop.q$ modulo M and that `xhpr` represents `top` modulo N according to the invariants:

(Kq0) $\quad$ `xnr` $=$ `top` **mod** $M \quad \wedge \quad$ `xhpr` $=$ `top` **mod** N ,
(Kq1) $\quad nr.q = ptop.q$ **mod** $M \quad \wedge \quad hpr.q = ptop.q$ **mod** N .

Preservation of (Kq0) at 39 follows from (Jq0). Preservation of (Kq1) at 12 and 15 follows from (Kq0). If M is a multiple of N, it follows from these invariants that we can eliminate `xhpr` and the private variables hpr. For the special case $M = 2N$, this elimination gives the original algorithm of [JP05].

The auxiliary variable $ptop.q$ serves to express the delay of process q. The relevant invariants are

(Kq2) $\quad ptop.q \leq$ `top` ,
(Kq3) $\quad ptop.q =$ `top` $\quad \equiv \quad q \in$ `xSet` .

Preservation of (Kq2) is trivial. Preservation of (Kq3) at 39 follows from (Kq2).

Array `bank` serves as a temporary protector for ind, according to the invariant

(Kq4) $\quad ptop.q <$ `top` $< ptop.q + M \quad \Rightarrow \quad$ `bank`$[nr.q] = ind.q$.

Preservation of (Kq4) at 33 follows from (Jq4), (Kq0), and (Kq1), at 39 from (Jq0), (Jq5), and (Kq3). We now claim the interference precluding invariant

(Kq4A) $\quad$ `top` $< ptop.q + M \quad \Rightarrow \quad ind.q \neq$ `mb`$.r$

This predicate follows from the invariants obtained above. To prove this, we assume `top` $< ptop.q + M$ and $ind.q =$ `mb`$.r$ and we derive a contradiction. By (Kq2), we have $ptop.q <$ `top` or $ptop.q =$ `top`. In the first case, (Kq4) and (Iq1) together imply $nr.q =$ `xnr`. By (Kq1) and (Kq0), this yields $ptop.q \equiv$ `top` (mod M). Since $ptop.q$ and `top` differ less than M, this gives $ptop.q =$ `top`, a contradiction. In the second case, (Kq3) and (Jq0) give $ind.q =$ `xind`, so that (Iq4) gives a contradiction.

Concerning the safe variables `buf`$[ind]$ read in line 13, we claim the invariant

(Mq4) $\quad pc.q = 13 \ \wedge$ `top` $< ptop.q + M \quad \Rightarrow \quad$ `buf`$[ind.q] =$ `hist`$(ptop.q)$.

Preservation of (Mq4) at 21 and 37 follows from (Kq4A). At 12 and 15, it follows from (Mq0), at 39 from (Kq2).

In view of the conditional jump at 14, we claim the invariant

(Mq5) $\quad pc.q = 14 \ \wedge$ `aux`$[q] \quad \Rightarrow \quad val.q =$ `hist`$(ptop.q)$.

This predicate is preserved at 39 because of (Kq2). Preservation at 13 follows from (Mq4) and the new postulate

(Kq5A) $\quad pc.q = 13 \ \wedge$ `aux`$[q] \quad \Rightarrow \quad$ `top` $< ptop.q + M$.

In order to prove (Kq5A), we strengthen it somewhat. The point is that, if process p remains long enough at line 13, some process will "help it" and make `aux`$[p]$ false in line 35. This is expressed in the invariant

(Kq5) $\quad pc.q = 13 \ \wedge$ `aux`$[q] \ \wedge \ ptop.q < n \ \wedge \ n$ **mod** $N = q \quad \Rightarrow \quad$ `top` $\leq n$.

Using $N < M$ and some elementary arithmetic one can show that (Kq5) indeed implies (Kq5A). For preservation of (Kq5) at 39, we first observe that `top` **mod** $N = q$ implies `xhpr` $= q$ because of (Kq0a). Therefore, preservation of (Kq5) at 39 follows from (Kq0a), (Kq3), and the additional invariant

(Kq6) $pc.q \in \{37, 38, 39\} \ \wedge \ q \in \texttt{xSet} \ \wedge \ pc.\texttt{xhpr} = 13 \ \wedge \ \texttt{aux}[\texttt{xhpr}]$
$\Rightarrow \ \texttt{xhpr} \in \texttt{xSet}$.

Preservation of (Kq6) at 34 follows from (Jq0), and at 35 from (Jq0) and the new invariant

(Kq7) $pc.q = 35 \ \wedge \ q \in \texttt{xSet} \ \wedge \ pc.\texttt{xhpr} = 13 \ \wedge \ \texttt{aux}[\texttt{xhpr}]$
$\Rightarrow \ q \in \texttt{heSet}[\texttt{xhpr}] \ \vee \ \texttt{xhpr} \in \texttt{xSet}.$

Preservation of (Kq7) at 34 follows from (Jq0).

8 The Synchronization Point

It still remains to decide how the number *ll* is chosen in line 12 of specification *ELSV* of Section 3. Procedure *LLmul*(p) contains three assignments to *val*.p, but we concentrate on line 18. There, process p reads the buffer at an index provided via line 14 by some other process in line 35. In this situation, we need to prove that $\texttt{buf}[bb.p] = hist(ll)$ for some number *ll* with $start.p \leq ll \leq \texttt{top}$. This number *ll* should be determined as the value of `top` at some moment during the execution of the "helping" process.

One might expect that *ll* should get the value of `top` when the helping process executes line 35 successfully. This should be justified by the conjectured invariant

(?) $pc.q = 35 \ \wedge \ q \in \texttt{heSet}[hpr.q] \ \Rightarrow \ q \in \texttt{xSet}$.

This conjecture, however, is not valid, as is shown by the following scenario. Assume that p_0 and p_1 are at lines 34 and 39, respectively, with $p_0, p_1 \in \texttt{xSet}$ both. Put $p_2 = \texttt{xhpr}$. Assume that p_2 executes 11 and 12. Then p_0 can execute 34 and enter 35 with $p_0 \in \texttt{heSet}[p_2]$. Then p_1 can execute 39 and remove p_0 from `xSet`, thus violating the conjectured invariant.

This shows that line 35 is not the synchronization point in the case of "helping". Indeed, the paper [JP05] correctly takes line 34 as the synchronization point we are looking for.

We thus give all processes private history variables *syn* and *ll*, and we use an array `htop` of history variables to virtually communicate the value of *syn* to the process that is being helped. We add $syn := \texttt{top}$ to line 34, and give $\texttt{htop}[hpr.p]$ the value of *syn* in the **then** branch of 35. We add $ll := \texttt{top}$ to lines 12 and 15, and $ll := \texttt{htop}[p]$ to 18.

```
34:     bb := hind[hpr] ; add(p, heSet[hpr]) ; syn := top ;
        if ¬ aux[hpr] ∨ p ∉ xSet then goto 37 ;
35:     if p ∈ heSet[hpr] then
           aux[hpr] := false ; hind[hpr] := mybuf ;
           heSet[hpr] := ∅ ; mybuf := bb ;
           htop[hpr] := syn end ;
12: 15: ind := xind ; nr := xnr ; hpr := xhpr ; add(p, xSet) ; ll := ptop := top ;
18:     val := buf[bb] ; ll := htop[p] ; goto 20 .
```

These extensions of the code clearly do not endanger the invariants obtained before. We now postulate the additional invariants

(Lq0) $ll.q \leq \texttt{top}$,
(Lq1) $\texttt{htop}[q] \leq \texttt{top}$,
(Lq2) $syn.q \leq \texttt{top}$,
(Lq3) $start.q \leq \texttt{top}$,
(Lq4) $pc.q \in \{13 \ldots 18\} \ \Rightarrow \ ll.q = ptop.q$.

Indeed, preservation of (Lq0) follows from (Lq1) at 17; preservation of (Lq1) follows from (Lq2) at 35. Preservation of (Lq2), (Lq3) and (Lq4) is trivial.

We now reach the invariants that will be used to relate the implementation with the specification *ESLV* of Section 3. The first of these invariants is

(Mq6) $\quad pc.q \in \{19, 20, 21\} \quad \Rightarrow \quad val.q = \texttt{hist}(ll.q).$

The proof of (Mq6) is based on:

(Mq2A) $\quad pc.q = 17 \ \wedge \ q \in \texttt{xSet} \quad \Rightarrow \quad val.q = \texttt{hist}(ll.q)$,
(Mq7A) $\quad pc.q = 18 \quad \Rightarrow \quad \texttt{buf}[bb.q] = \texttt{hist}(\texttt{htop}[q])$.

Indeed, preservation of (Mq6) follows at 14 from (Mq5) and (Lq4), at 17 from (Mq2A), at 18 from (Mq7A), and at 39 from (Lq0).

Predicate (Mq2A) follows from (Mq2), (Kq3), and (Lq4). Predicate (Mq7A) follows from (Jq1), (Jq9), and (Mq7), where (Mq7) is given by

(Mq7) $\quad pc.q \in \{12 \ldots 18\} \ \wedge \ \neg\, \texttt{aux}[q] \quad \Rightarrow \quad \texttt{buf}[\texttt{hind}[q]] = \texttt{hist}(\texttt{htop}[q])$,
(Mq8) $\quad pc.q = 35 \quad \Rightarrow \quad \texttt{buf}[mybuf.q] = \texttt{hist}(syn.q)$.

Preservation of (Mq7) uses (Mq8) at 35, (Iq0) at 21 and 37, (Lq1) at 39. Preservation of (Mq8) uses (Mq3) at 34, (Iq0) at 21 and 37, (Lq2) at 39.

In view of the specification, we also claim the invariant

(Lq5) $\quad q \in \texttt{xSet} \quad \equiv \quad ll.q = \texttt{top}$.

The predicate is preserved at 39 because of (Lq0), and at 17 because of (Lq0) and the new invariant

(Lq6) $\quad pc.q \in \{16, 17, 18\} \quad \Rightarrow \quad \texttt{htop}[q] \leq ll.q$

For the proof of (Lq6), we postulate the additional invariants

(Lq7) $\quad pc.q = 12 \quad \vee \quad start.q \leq ll.q$,
(Lq8) $\quad pc.q \in \{12 \ldots 18\} \ \wedge \ \neg\, \texttt{aux}[q] \quad \Rightarrow \quad start.q \leq \texttt{htop}[q]$,
(Lq9) $\quad pc.q = 35 \ \wedge \ q \in \texttt{heSet}[hpr.q] \quad \Rightarrow \quad start.(hpr.q) \leq syn.q$.

Preservation of (Lq6) at 15 and 35 follows from (Lq1), (Jq1), and (Lq7). Predicate (Lq7) is preserved at 12 and 15 because of (Kq2) and (Lq3), at 17 because of (Jq1) and (Lq8). Predicate (Lq8) is preserved at 35 because of (Lq9). Preservation of (Lq9) follows at 11 from (Jq1) and (Jq7), at 34 from (Lq3).

9 Forward Simulation

In this section, we show that the algorithm implements LL/SC according to the second specification of Section 3. We assume that the procedures *LLe*, *SCe*, and *VLe* return to location 50, just as the concrete procedures. So, location 50 and its command as given in Section 4 also serve in the abstract specification. We have two state spaces, the concrete state space of Section 4 and the abstract state space

$$\begin{aligned} &AbState = [\#\quad \texttt{hist} : \mathbb{N} \to Item\,;\ \texttt{top} : \mathbb{N}\ , \\ &\quad start, ll, pc : Process \to \mathbb{N}\,;\ val, arg : Process \to Item\ , \\ &\quad result : Process \to \mathbb{B}\quad \#]\ . \end{aligned}$$

We let x, y range over the concrete state space and u, v range over the abstract state space. If w is any field of the concrete or abstract state space, we write $x.w$ (etc.) for the value of w in state x. Let $step(x, y)$ mean that the concrete algorithm can do a step from state x to state y (or $x = y$), and let $astep(u, v)$ mean that the abstract specification can to a step from u to v or that $u = v$.

Outside procedure *LLmul*, we need to regard the private variables *val* and *ll* as visible, but inside this procedure they may (and do) hold uninteresting temporary values. This is a complication that precludes the use of a refinement mapping cf. [AL91]. Instead we use a forward simulation relation [HHS86,LV95,Mil71], which allows us to ignore the values of these variables during *LLmul*.

We therefore define the relation F between the concrete state space and the abstract state space by $(x, u) \in F$ iff

$$\begin{array}{l} x.\mathtt{hist} = u.\mathtt{hist} \ \wedge \ x.\mathtt{top} = u.\mathtt{top} \ \wedge \ x.arg = u.arg \\ \wedge \quad x.result = u.result \ \wedge \ x.start = u.start \\ \wedge \quad (\forall\, q : u.pc.q = g(x.pc.q) \\ \qquad \wedge \quad (x.pc.q > 21 \ \Rightarrow \ x.ll.q = u.ll.q \ \wedge \ x.val.q = u.val.q)\ ) \ , \end{array}$$

where $g(n) = 12$ for $12 \leq n \leq 21$, $g(n) = 32$ for $32 \leq n \leq 39$, and else $g(n) = n$. It easily follows that every concrete initial state x_0 allows an abstract intial state u_0 with $(x_0, u_0) \in F$.

Let Mq stand for the set of concrete states where the conjunction of (Mq6), (Lq0), and (Lq7) holds, and let $Lq5$ stand for the set where (Lq5) holds.

Lemma 1. *If $step(x, y)$ and $x \in Mq \cap Lq5$ and $(x, u) \in F$, there is an abstract state v with $astep(u, v)$ and $(y, v) \in F$.*

Proof. We define $step(p, n, x, y)$ to mean that in concrete state x, process p has $x.pc.p = n$ and p can do a step such that y is the new state. Similarly, $astep(p, n, u, v)$ means that in abstract state u, process p has $u.pc.p = n$ and p can do a step such that v is the new abstract state. We now verify that, for $(x, u) \in F$,

$$\begin{array}{l} step(p, 21, x, y) \ \wedge \ x \in Mq \quad \Rightarrow \quad \exists\, v : (y, v) \in F \wedge astep(p, 12, u, v) \ , \\ step(p, 39, x, y) \ \wedge \ x \in Lq5 \quad \Rightarrow \quad \exists\, v : (y, v) \in F \wedge astep(p, 32, u, v) \ , \\ step(p, 45, x, y) \ \wedge \ x \in Lq5 \quad \Rightarrow \quad \exists\, v : (y, v) \in F \wedge astep(p, 45, u, v) \ , \\ step(p, n, x, y) \ \wedge \ n \in \{11, 50\} \quad \Rightarrow \quad \exists\, v : (y, v) \in F \wedge astep(p, n, u, v) \ , \\ step(p, n, x, y) \ \wedge \ n \notin \{11, 21, 39, 45, 50\} \quad \Rightarrow \quad (y, u) \in F \ \square \end{array}$$

This result shows that, under assumption of the invariants, every step of the concrete algorithm can be mimicked by a step of the abstract algorithm. We therefore define *Inv* to be the set of states where all invariants listed above hold, and let F' be the set of pairs $(x, u) \in F$ with $x \in Inv$. Then $step(x, y)$ and $(x, u) \in F'$ implies that the existence of v with $astep(u, v)$ and $(y, v) \in F'$. Since the procedures have no loops, we need not bother about progress conditions. We thus have:

Theorem 1. *Relation F' is a forward simulation from the concrete system to the abstract specification.*

This concludes the proof that the algorithm of Jayanti and Petrovic is correct. As announced above, we constructed and verified this proof with the proof assistant PVS [OSRSC01]. The proof script is available at [Hes06].

10 The Pure LL/SC Algorithm

As announced at the end of Section 2, we here treat the variation of the algorithm in which the variables `xx`, `help`$[p]$, and `bank`$[k]$ are accessed only via LL, VL, and SC. Firstly, line 11 is implemented by means of LL followed by SC as indicated there. Therefore, line 11 is split into two atomic commands for which we use the line numbers 10 and 11. Secondly, the atomic read operations in the lines 20 and 38 are replaced by LL operations such that the acting process enters its identifier in the corresponding set. The lines 11, 20, and 38 are thus replaced by

10: $start := \texttt{top}$; $add(p, \texttt{heSet}[p])$;
11: **if** $p \in \texttt{heSet}[p]$ **then**
$\texttt{aux}[p] := true$; $\texttt{hind}[p] := mybuf$; $\texttt{heSet}[p] := \emptyset$;
20: $mybuf := \texttt{hind}[p]$; $add(p, \texttt{heSet}[p])$;
38: $bb := \texttt{bank}[(nr + 1) \textbf{ mod } M]$; $add(p, \texttt{baSet}[(nr + 1) \textbf{ mod } M])$.

Moreover, in line 50, we replace **goto** 11 by **goto** 10.

This modification only directly threatens correctness of the invariants (Iq0), (Iq1), (Iq4), (Mq7), (Lq7) and (Lq8). Since the assignment to *start* has moved to line 10, we need to modify (Lq7) into

(Lq7') $\quad pc.q = 11 \quad \vee \quad pc.q = 12 \quad \vee \quad start.q \leq ll.q$.

The other invariants can be retained, but in order to preserve them at line 11 we need to postulate the additional invariant

(Lq10) $\quad pc.q = 11 \quad \Rightarrow \quad q \in \texttt{heSet}[q]$.

Preservation of (Lq10) at 35 follows from (Jq1) and (Jq7). Finally, in the forward simulation, the role of location 11 is taken over by 10. After these changes, everything goes through as before. Indeed, it was easy to adapt the PVS proof script of the original algorithm to the pure alternative. When making the alternative, we realized the relevance of the new invariant (Lq10), but we had missed the need of (Lq10) for preservation of (Iq0), (Iq1), and (Iq4) at 11. This came up while adapting the proof script for PVS.

11 Mixed Conclusions

We have given a refinement proof of the wait-free implementation [JP05] of LL, SC and VL on a multiword variable by means of LL, SC and VL on single words and $3N$ safe variables, and have obtimized it to an implementation that only uses $2N + 1$ safe variables. Since there are cheaper and more powerful implementations of arrays of lock-free multiword variables, we regard the present result primarily of theoretical interest.

References

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82:253–284, 1991.

[DHLM04] S. Doherty, M. Herlihy, V. Luchangco, and M. Moir. Bringing practical lock-free synchronization to 64-bit applications. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 31–39, 2004.

[GH04] H. Gao and W.H. Hesselink. A formal reduction for lock-free parallel algorithms. In R. Alur and D.A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004*, volume 3114 of *LNCS*, pages 44–57. Springer, 2004.

[Hes05] W.H. Hesselink. Eternity variables to prove simulation of specifications. *ACM Trans. on Comp. Logic*, 6:175–201, 2005.

[Hes06] W.H. Hesselink. A refinement proof of a multiword LL/SC object. Report and PVS files available at `www.cs.rug.nl/~wim/pub/mans.html`, 2006.

[HHS86] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *ESOP 86*, volume 213 of *LNCS*, pages 187–196, New York, 1986. Springer.

[JP05] P. Jayanti and S. Petrovic. Efficient wait-free implementation of multiword LL/SC variables. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS), June 2005*, pages 59–68. IEEE, 2005.

[Lam86] L. Lamport. On interprocess communication. Parts I and II. *Distr. Comput.*, 1:77–101, 1986.

[LV95] N. Lynch and F. Vaandrager. Forward and backward simulations, part I: untimed systems. *Inf. Comput.*, 121:214–233, 1995.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd Int. Joint Conf. on Artificial Intelligence*, pages 481–489. British Comp. Soc., 1971.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Inf.*, 6:319–340, 1976.

[OSRSC01] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference*, 2001. `http://pvs.csl.sri.com`