Formal Description Techniques and Protocol Specification, Testing, and Verification,
FORTE/PSTV'96, Kaiserslauten, Germany, 8–11 October 1996.

Computer-Assisted Verification of an Algorithm for Concurrent Timestamps

Tsvetomir P. Petrov, Anna Pogosyants, Stephen J. Garland,
Victor Luchangco, and Nancy A. Lynch
MIT Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139

This paper is dedicated to the memory of one of its authors, Anna Pogosyants, who was working on this paper and her Ph. D. thesis when she was killed in an automobile accident in December 1995.

Abstract

A formal representation and machine-checked proof are given for the Bounded Concurrent Timestamp (BCTS) algorithm of Dolev and Shavit. The proof uses invariant assertions and a forward simulation mapping to a corresponding Unbounded Concurrent Timestamp (UCTS) algorithm, following a strategy developed by Gawlick, Lynch, and Shavit. The proof was produced interactively, using the Larch Prover.

Keywords

Verification, validation and testing; tools and tool support; Larch; input/output automata; concurrent timestamps

1 INTRODUCTION

In this paper, we describe a computer-assisted verification, using the Larch Prover (Garland and Guttag, 1991), of one of the most complicated algorithms in the distributed systems theory literature: the Bounded Concurrent Timestamp (BCTS) algorithm of Dolev and Shavit (1989). This algorithm runs in the single-writer, multi-reader, read/write shared memory model. The verified algorithm is a slight simplification, due to Gawlick, Lynch, and Shavit (1992), of the original algorithm. It uses atomic snapshots of the shared memory, which Afek *et. al.* (1990) showed can be implemented in an efficient and fault-tolerant manner using read and write operations. Hence the simplification is not a

Research supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-92-J-1795 and by Hanscom Air Force Base under contract F19628-95-C-0018, by the National Science Foundation under grants CCR-9504248 and CCR-9225124, and by the Air Force Office of Scientific Research and the Office of Naval Research under contract F49620-94-1-0199.

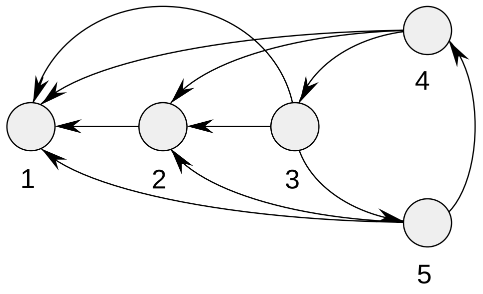

Figure 1 Domain T_2 of timestamps for two users, with pairwise ordering relationships

significant change in the original algorithm, though its introduction does remove some of the complexities of the proof.

The algorithm allows n users to (asynchronously) submit data values in *label* operations, and to request information about recent values submitted by all users in *scan* operations. Roughly speaking, a *scan* operation returns the most recent value submitted by each user, together with a total order corresponding to the temporal order of those submissions. Because operations can be concurrent, this temporal order is not always clearly determined, so some care is needed in defining what it means for the algorithm to be correct.

Originally, the correctness conditions were defined axiomatically. In this paper, we define them instead in terms of the behaviors exhibited by another, simpler algorithm—the Unbounded Concurrent Timestamp (UCTS) algorithm of Gawlick, Lynch, and Shavit (1992). An easy proof of the correctness of UCTS was given by Gawlick (1992).

UCTS maintains a global data vector with one component for each user and with each component accompanied by a nonnegative real-valued timestamp.* A *scan* operation reads the vector (all at once) and uses the timestamps to determine the total order, breaking ties by user identifiers. A *label* operation reads the vector (all at once), determines the largest real-valued timestamp, and chooses any greater real number as a new timestamp. (There is one exception: if the choosing process itself is already the "winner," it simply retains its previous timestamp.) Then, *in a separate step*, it attaches this new timestamp to the value to be written, and writes the pair into its component of the global vector. Other operations may interleave between the reading and writing step of a *label* operation.

UCTS is interesting because it can be used as a building block in the solution of several distributed system problems, including first-come first-serve mutual exclusion (Lamport, 1974) and the construction of a multi-writer multi-reader atomic register from single-writer multi-reader registers (Vitanyi and Awerbuch, 1986).

The Dolev-Shavit BCTS algorithm implements UCTS (i.e., its behaviors are allowed by UCTS), but uses a bounded domain of timestamps rather than the unbounded set of nonnegative reals. The particular domain used is a nested graph, nested to depth $n-1$, where n is the number of users. The basis of the domain is the 5-node digraph T_2 shown in Figure 1, in which arrows indicate which nodes come before which in the ordering. Note that this ordering is not a partial order, since it is not transitive—it only gives pairwise

*Any unbounded dense order, e.g., the nonnegative rationals, will work just as well.

ordering relationships between nodes. The timestamp domain T_n for n users consists of T_2, with each node replaced by another instance of T_2, nested to a depth of $n-1$. Pairwise ordering relationships between nodes in T_n are determined from those in T_2, based on the smallest instance of T_2 that contains both nodes.

The distributed algorithms community considers BCTS to be very complicated. The correctness proof by Dolev and Shavit (1989), based on ordering relations defined by Lamport (1986), is long, detailed, and hard to understand. In fact, the algorithm and proof have been considered so complicated that there have been at least two serious attempts to devise simpler algorithms (Israeli and Li, 1987; Dwork and Waarts 1992). Note that the difficulty of the proof is entirely in the *safety* properties—liveness is obvious.

Gawlick, Lynch, and Shavit (1992) gave a correctness proof for BCTS that has a nicer structure than the original proof. Their proof is based on the input/output (I/O) automaton model (Lynch and Tuttle, 1987; Lynch, 1996) and uses a set of invariant assertions and a forward simulation mapping (Lynch and Vaandrager, 1991) from BCTS to UCTS. The mapping and most of the invariants provide useful intuition about the algorithm and make the proof much easier to understand. However, the proofs of the invariants and the mapping are still rather long and detailed (about 20 pages in Gawlick, 1992). They consist of many cases and large numbers of algebraic substitutions. The case arguments are so detailed and delicate, in fact, that they did not seem to us to be sufficiently convincing.

Because we were unsatisfied with the prior proof, and because the BCTS algorithm is so fundamental, we undertook the task of verifying BCTS using the Larch Prover (Garland and Guttag, 1991). Because of the complexity of the algorithm and proof, we believed this would "push the limits" of current theorem-proving technology. We made one modification to the algorithm: to avoid clutter in the proofs, we suppressed the data values and worked only with the timestamps.[†] We have completed the proof, and describe our results and experiences in the rest of this paper.

Briefly, our proof is based on that of Gawlick (1992), using essentially the same invariants and simulation mapping. Our initial work involved formalizing many concepts (data types, automata,...) that were previously defined only semi-formally. In doing this, we discovered and resolved several small ambiguities in the earlier paper. Our formalization led to an overall clarification and improvement of the earlier definitions.

Our proof meshes nicely with the earlier proof, filling in gaps in the reasoning and verifying the correctness of all the steps. Our proof contains a considerable amount of advice from the user to the Larch Prover, but also many instances where the Larch Prover filled in steps that were done by hand in the earlier proof. We discovered no serious errors in the earlier proof, but did discover some significant missing steps, including a (simple) missing invariant.

We constructed the proof without major difficulties. In fact, an MIT freshman with no knowledge of automated theorem-proving or of distributed algorithms completed an initial version of the entire proof in only 8 weeks. Subsequent weeks were spent polishing and reorganizing the proof for maximum clarity and generality, and analyzing the specification for potential inconsistencies.

The contributions of this paper are (1) a formal proof for an important distributed

[†]The values are merely "piggybacked" on the timestamps anyhow, so this should not be a significant modification. Still, we plan to integrate the data values and re-run our proof.

```
start(s) ⇔ ∀ i ∀ j (   s[i].pc = nil  ∧  s[i].op = nil
                     ∧  s[i].t = 0    ∧  s[i].nt = 0   ∧  (s[i].ord)[j] = j);

enabled(s, beginscan[i]) ⇔ i ≠ 0;
effect(s, beginscan[i], s') ⇔
  s'[i].op = scan  ∧  s'[i].pc = snap  ∧  almostsame(s, s', i);

enabled(s, snap[i]) ⇔ i ≠ 0  ∧  s[i].pc = snap;
effect(s, snap[i], s') ⇔
    (if s[i].op = scan
        then   ∀ j ∀ k (   (s'[i].ord)[j] < (s'[i].ord)[k]
                        ⇔ s'[j].t < s'[k].t  ∨  (s'[j].t = s'[k].t  ∧  j < k))
             ∧  s'[i].pc = endscan  ∧  same_nt(s, s', i)
        else    s'[i].ord = s[i].ord  ∧  s'[i].pc = update
             ∧  newlabel(s, i, s'[i].nt))
  ∧  ∀ j (i ≠ j ⇒ s[j] = s'[j])  ∧  same_t(s, s', i)  ∧  same_op(s, s', i);

enabled(s, endscan(o)[i]) ⇔ i ≠ 0  ∧  s[i].pc = endscan  ∧  s[i].ord = o;
effect(s, endscan(o)[i], s') ⇔
  s'[i].op = nil  ∧  s'[i].pc = nil  ∧  o = s'[i].ord  ∧  almostsame(s, s', i);

enabled(s, beginlabel[i]) ⇔ i ≠ 0;
effect(s, beginlabel[i], s') ⇔
  s'[i].op = label  ∧  s'[i].pc = snap  ∧  almostsame(s, s', i);

enabled(s, update[i]) ⇔ i ≠ 0  ∧  s[i].pc = update;
effect(s, update[i], s') ⇔
     s'[i].t = s[i].nt  ∧  s'[i].pc = endlabel  ∧  ∀ j (i ≠ j ⇒ s'[j] = s[j])
  ∧  same_nt(s, s', i)  ∧  same_ord(s, s', i)  ∧  same_op(s, s', i);

enabled(s, endlabel[i]) ⇔ i ≠ 0  ∧  s[i].pc = endlabel;
effect(s, endlabel[i], s') ⇔
  s'[i].op = nil  ∧  s'[i].pc = nil  ∧  almostsame(s, s', i)
```

Figure 2 Specifications for CTS actions

algorithm, and (2) a convincing demonstration of the practicality of using automated theorem-provers in verifying full-scale distributed algorithms.

2 THE BCTS ALGORITHM

We model each of BCTS and UCTS as a shared memory I/O automaton (Lynch, 1996), with one process for each user $i \in \{1, \ldots, n\}$.[‡] Each automaton has input actions $beginscan_i$ and $beginlabel_i$, output actions $endscan_i$ and $endlabel_i$, and internal actions $snap_i$ and $update_i$ for each $i \in \{1, \ldots, n\}$. In fact the two algorithms are sufficiently similar that we formalize them both in terms of a generic concurrent timestamp algorithm, CTS (specified

[‡]For technical reasons related to the Larch type system, we have actually included a process 0; however, it has no defined steps.

```
UCTS (U, Label): trait
  includes CTS(U, Label), RationalOrder(Label)
  asserts with i, j: UID, s: States[U], l: Label
    ∃ i (s[i].t = tmax(s)) ∧ ∀ i (s[i].t ≤ tmax(s));
    tmax(s) = s[imax(s)].t ∧ ∀ j (tmax(s) = s[j].t ⇒ j ≤ imax(s));
    newlabel(s, i, l) ⇔ (if i = imax(s) then l = tmax(s) else tmax(s) < l)
```

Figure 3 Specification for UCTS

using Larch in Figure 2), which is parameterized by the sort *Label* used for timestamps, an appropriate ordering and an appropriate definition of the *newlabel* relation.

The state of CTS has the following components, for each i:

- t_i: The label currently used as a timestamp for user i, initially 0.
- nt_i: A label (satisfying the *newlabel* predicate) to be used as a new timestamp for i, initially 0.
- ord_i: An array giving ranks of each process in timestamp order.
- pc_i: The currently enabled non-input action, initially *nil*.
- op_i: the operation of which this action is a part, initially *nil*.

The t_i are shared variables, each t_i writable by process i and readable by all processes; nt_i, ord_i, pc_i, and op_i are private variables of process i.

The specification of CTS (Figure 2) provides, for each action, an *enabling predicate* on the state describing when the action can occur, and an *effect predicate* relating the pre- and post-states when the action occurs. The state initialization is also described by a predicate. The *begin* actions just set the *op* components appropriately and set the *pc* components to *snap*. A $snap_i$ action within a *scan* operation sets ord_i to the total ordering of process indices given by the timestamps; ties are broken by process index. (The relation $<$ on the ord_i components is the usual ordering of the integers; the relation $<$ on the t_i components is the ordering assumed to be defined on the sort *Label*.) Then it sets pc_i to *endscan*. A $snap_i$ within a *label* operation, on the other hand, places in nt_i a new timestamp satisfying the *newlabel* predicate, then sets pc_i to *update*. An $update_i$ sets the shared component t_i equal to the private component nt_i. The *end* actions clean up the state; $endscan_i$ also returns the current ord_i.

Figure 2 does not contain the complete Larch specification (or *trait*) for CTS. The rest consists mostly of type definitions, function declarations, and definitions of predicates like *almostsame* and *same_nt*, which are used to specify that other state components are unchanged. This specification incorporates by reference other Larch traits, such as IOAutomaton, which provide general definitions.

Figure 3 contains the specification of a concurrent timestamp automaton, UCTS, that uses an unbounded set of labels (namely, the nonnegative rational numbers) for timestamps. Here, the definition of the *newlabel* predicate is particularly simple, since there is always a label that is greater than the maximum of the labels t_j.

Figures 4 and 5 define the finite set of labels used as timestamps by BCTS. These labels are lexicographic sequences of $n-1$ symbols from the five-symbol alphabet shown

```
Symbol: trait
  introduces
    s1, s2, s3, s4, s5:               → Symbol,
    NS:                Symbol         → Symbol,
    __ <| __:          Symbol, Symbol → Bool
  asserts
    sort Symbol generated freely by s1, s2, s3, s4, s5
    with t, t1, t2: Symbol
      NS(s1) = s2; NS(s2) = s3; NS(s3) = s4; NS(s4) = s5; NS(s5) = s3;
      t <| NS(t); s1 <| s3; s1 <| s4; s1 <| s5; s2 <| s4; s2 <| s5;
      ¬(t1 <| t2 ∧ t2 <| t1);
```

Figure 4 Axioms for symbols

```
TimeStamp (Symbol, Label, Index): trait
  includes ZeroToN(Index, n), Symbol, Array(Label, Index, Symbol)
  introduces
    0:                               → Label
    __ < __:     Label, Label        → Bool   % lexicographic order
    sameThrough: Label, Index, Label → Bool
    sameBefore:  Label, Index, Label → Bool
    nextlabel:   Label, Index        → Label
    nextlabel3:  Label, Label, Index → Bool
    inCycle:     Label, Label, Index → Bool
    trans:       Label, Label, Label → Bool   % transitivity
    __≤__:       Label, Label        → Bool
  asserts with t1, t2: Symbol, l, l1, l2, l3: Label, h, p: Index
    sameThrough(l1, h, l2) ⇔ ∀ p (p ≤ h ⇒ l1[p] = l2[p]);
    sameBefore(l1, h, l2)  ⇔ ∀ p (p <  h ⇒ l1[p] = l2[p]);
    l1 < l2 ⇔ ∃ h (sameBefore(l1, h, l2) ∧ l1[h] <| l2[h]);
    l1 ≤ l2 ⇔ l1 < l2 ∨ l1 = l2;
    0[p] = s1;
    trans(l1, l2, l3)  ⇔ l1 < l2 ∧ l2 < l3 ⇒ l1 < l3;
    inCycle(l1, l2, h) ⇔ sameBefore(l1, h, l2) ∧ s2 <| l1[h];
    nextlabel3(l1, l2, h) ⇔   h ≠ 0 ∧ sameBefore(l1, h, l2)
                            ∧ l1[h] = NS(l2[h]) ∧ ∀ p (h < p ⇒ l1[p] = s1);
    h ≠ 0 ⇒ nextlabel3(nextlabel(l2, h), l2, h);
```

Figure 5 Axioms for timestamps

```
BCTS (B, n): trait
  includes CTS(B, Label), TimeStamp(Symbol, Label, UID), IndexSet(UID, n)
  introduces
    agree: States[B], Label, UID → FiniteSet[UID]
    numi:  States[B], UID, UID   → UID
    fulli: States[B], UID, UID   → Bool
    full:  States[B], UID        → UID
  asserts with l: Label, s: States[B], h, i, j, k: UID
    ∀ i ∀ j ∀ k trans(s[i].t, s[j].t, s[k].t)
      ⇒ ∃ i (s[i].t = tmax(s)) ∧ ∀ i (s[i].t ≤ tmax(s));
    ∀ i ∀ j ∀ k trans(s[i].t, s[j].t, s[k].t)
      ⇒ tmax(s) = s[imax(s)].t ∧ ∀ j (tmax(s) = s[j].t ⇒ j ≤ imax(s));
    j ∈ agree(s, l, h) ⇔ j ≠ 0 ∧ sameThrough(s[j].t, h, l);
    numi(s, i, h) = size(agree(s, tmax(s), h) - {i});
    fulli(s, i, h) ⇔ (n - h) ≤ numi(s, i, h);
    fulli(s, i, full(s, i)) ∧ ∀ j (j < full(s, i) ⇒ ¬fulli(s, i, j));
    newlabel(s, i, l) ⇔
      l = (if i = imax(s) then s[i].t else nextlabel(tmax(s), full(s, i)))
```

Figure 6 Specification for BCTS

in Figure 1.[§] Figure 6 uses these definitions to define a bounded concurrent timestamp automaton, BCTS. In order to define the *newlabel* predicate here, we first define $agree(l, h)$ as the set of all process indices whose timestamps agree with l in the first h components. Then, assuming that there is some maximum timestamp t_{max}, and letting i_{max} be the largest process index with $t = t_{max}$, we define the unique label l satisfying *newlabel* as follows. It is t_i if $i = i_{max}$. Otherwise, we let h be the minimum index such that $size(agree(t_{max}, h) - \{i\}) \geq n - h$, and we let l be the timestamp that agrees with t_{max} in the first $h-1$ components, whose h component is the graph-successor of the h component of t_{max}, and whose other components are all s_1.

A technicality: Note that this definition assumes that there is some maximum timestamp. This is not the case in general, because $<$ is only defined pairwise and is not a true partial order. Our conventions say that, if there is no such maximum, then t_{max} and i_{max} are defined arbitrarily. In places where the definition of *newlabel* is used, however, state invariants will show that for all i, j, and k, $t_i < t_j \wedge t_j < t_k \Rightarrow t_i < t_k$, which implies that t_{max} is defined.

3 MANUAL PROOF OF CORRECTNESS

The correctness proof in (Gawlick, 1992) proceeds by showing that there is some relation between the states of BCTS and those of UCTS that is a (multivalued) *forward simulation*, as defined in Figure 7. An easy proof by induction shows that the existence of such

[§]For technical reasons, we have actually formalized the labels as sequences of $n + 1$ symbols; however, positions 0 and n are not used (a fact that we verify).

```
SimulationMap (A1, A2, F): trait
  assumes IOAutomaton(A1), IOAutomaton(A2)
  introduces F: States[A1], States[A2] → Bool
  asserts with s, s': States[A1], u: States[A2], a: Actions[A1],
               alpha: StepSeq[A2]
    start(s) ∧ inv(s) ⇒ ∃ u (start(u) ∧ F(s, u));
    F(s, u) ∧ inv(s) ∧ inv(s') ∧ isStep(s, a, s') ⇒
       ∃ alpha (execFrag(alpha) ∧ first(alpha) = u ∧ F(s', last(alpha))
                  ∧ trace(alpha) = trace(a))
```

Figure 7 Requirements for a forward simulation relation

```
SimulationBU (B, U, n): trait
  includes BCTS(B, n), UCTS(U, ULabel)
  introduces F: States[B], States[U] → Bool
  asserts with b: States[B], u: States[U], i, j, k: UID
    F(b, u) ⇔
         ∀ i (   (b[i]).ord = (u[i]).ord ∧ (b[i]).op = (u[i]).op
              ∧ (b[i]).pc = (u[i]).pc)
       ∧ ∀ i ∀ j ((u[i]).t = (u[i]).nt ⇔ (b[i]).t = (b[i]).nt)
       ∧ ∀ i ∀ j (   i ≠ j
                     ⇒ (   ((b[i]).t  < (b[j]).t  ⇔ (u[i]).t  < (u[j]).t)
                         ∧ ((b[i]).t  = (b[j]).t  ⇔ (u[i]).t  = (u[j]).t)
                         ∧ ((b[i]).nt < (b[j]).nt ⇔ (u[i]).nt < (u[j]).nt)
                         ∧ ((b[i]).nt < (b[j]).t  ⇔ (u[i]).nt < (u[j]).t)
                         ∧ ((b[i]).t  < (b[j]).nt ⇔ (u[i]).t  < (u[j]).nt)))
```

Figure 8 Simulation relationship between BCTS and UCTS

a forward simulation implies that all behaviors of BCTS are also behaviors of UCTS (Lynch and Vaandrager, 1987) and hence that BCTS is correct.

Figure 8 defines the purported simulation relation, which basically says that the ordering relationships between occurrences of timestamps in the states of the two algorithms are the same.¶ The fact that this relation is a forward simulation follows from a list of invariants for BCTS, shown in Figure 9. Invariant *inv1* says that all "significant" triples of timestamp occurrences have consistent pairwise ordering relationships. Invariant *inv2* says that the t and nt components of process i_{max} are equal. Invariant *inv3* says that any nt greater than t_{max} must have been assigned as a result of a *nextlabel* operation. The other four are more technical; they are used in an inductive proof of invariants *inv1* through *inv3*.

The proof also uses several auxiliary lemmas about timestamps, the most interesting

¶The definition in Figure 8 actually adds two clauses to the one in (Gawlick, 1992). We needed these for our computer-assisted proof; we have not yet determined whether they can be eliminated.

```
inv1(s) ⇔
  ∀ i ∀ j ∀ k (
             trans(s[i].t, s[j].t, s[k].t)
          ∧  (k ≠ i  ∧  k ≠ j ⇒ trans(s[i].t,  s[j].t,  s[k].nt))
          ∧  (j ≠ i  ∧  j ≠ k ⇒ trans(s[i].t,  s[j].nt, s[k].t))
          ∧  (i ≠ j  ∧  i ≠ k ⇒ trans(s[i].nt, s[j].t,  s[k].t))
          ∧  (i ≠ j  ∧  i ≠ k ⇒ trans(s[i].t,  s[j].nt, s[k].nt))
          ∧  (j ≠ i  ∧  j ≠ k ⇒ trans(s[i].nt, s[j].t,  s[k].nt))
          ∧  (k ≠ i  ∧  k ≠ j ⇒ trans(s[i].nt, s[j].nt, s[k].t))
          ∧  trans(s[i].nt, s[j].nt, s[k].nt));
inv2(s) ⇔
  s[imax(s)].t = s[imax(s)].nt;
inv3(s) ⇔
  ∀ p (   tmax(s) < s[p].nt
      ⇒ ∃ h (s[p].nt = nextlabel(tmax(s), h)  ∧  h ≠ 0));
inv4(s) ⇔
  ∀ i ∀ h (   s[i].nt ≤ tmax(s)  ∧  sameThrough(s[i].t, h, tmax(s))
          ⇒ sameThrough(s[i].nt, h, tmax(s)));
inv5(s) ⇔
  ∀ i ∀ h (inCycle(s[i].nt, tmax(s), h) ⇒ sameBefore(s[i].t, h, tmax(s)));
inv6(s) ⇔
  ∀ i ∀ h (   s[i].nt = nextlabel(tmax(s), h)  ∧  h ≠ 0
          ⇒ (n - h) ≤ numi(s, i, pred(h)));
inv7(s) ⇔
  ∀ h (   tmax(s)[h] ≠ s1  ∧  h ≠ 0
      ⇒ (n - h) < size(agree(s, tmax(s), pred(h))));
```

Figure 9 Invariants preserved by BCTS

```
implies with l1, l2, l3: Label, h: Index
  ¬(l1 < l2  ∧  l2 < l1);
  l1 < l2  ∨  l1 = l2  ∨  l2 < l1;
  l1 < l2  ∧  l2 < l3  ∧  ¬(l1 < l3)
    ⇒ ∃ h (   sameBefore(l1, h, l2)  ∧  sameBefore(l2, h, l3)
           ∧  (   (l1[h] = s3  ∧  l2[h] = s4  ∧  l3[h] = s5)
               ∨  (l1[h] = s4  ∧  l2[h] = s5  ∧  l3[h] = s3)
               ∨  (l1[h] = s5  ∧  l2[h] = s3  ∧  l3[h] = s4)))
```

Figure 10 Lemmas about timestamps

of which are shown in Figure 10. The first two imply trichotomy: for any two labels l_1 and l_2, exactly one of the relations $l_1 < l_2$, $l_1 = l_2$, $l_1 > l_2$ is true. The third describes the circumstances in which a set of three labels can fail to be linearly ordered by $<$.

An example of a typical claim (about an $update_k$ step) and its manual proof appear in Figure 11.

Claim 6.4.6: If $k \neq b.i_{max}$ and $b.t_{max} \prec b.nt_k$ then $b'.t_{max} = b'.t_k$ and $b'.i_{max} = k$.

Proof. We proceed by showing that $b'.t_i \prec b'.t_k$ for all $i \neq k$. From the definition of t_{max} and the assumption that $b.t_{max} \prec b.nt_k$, we know that $b.nt_i \preceq b.t_{max} \prec b.nt_k$. Let $z = b.i_{max}$; then $b.t_z = b.t_{max}$ and $z \neq k$. Since $k \neq z$, $k \neq i$, and $b.t_z = b.t_{max}$, there exists a choice vector that includes the values $b.t_i, b.t_{max}$, and $b.nt_k$. Since $\text{TOT}(b) = true$, the values in this choice vector are totally ordered. Hence, $b.t_i \preceq b.t_{max} \prec b.nt_k$ implies that $b.t_i \prec b.nt_k$. As a result of the action $b.nt_k = b'.t_k$ and t_i does not change. Therefore, $b.t_i \prec b.nt_k$ implies that $b'.t_i \prec b'.t_k$. Hence $b'.t_{max} = b'.nt_k$. Since k is the only process index for which $b'.t_{max} = b'.t_k$, $b'.i_{max} = k$. □

Figure 11 Manual proof of Claim 6.4.6 (Gawlick, 1992)

4 COMPUTER-ASSISTED PROOF OF CORRECTNESS

Our computer-assisted proof was carried out using the Larch Prover (Garland and Guttag, 1991), an automated proof assistant for multisorted first-order logic that employs built-in knowledge about boolean operations and quantifiers, equational term rewriting, and proof techniques like case analysis and structural induction. Our proof follows Gawlick's (1992) closely, differing mainly in the following respects.

- It provides axioms for the underlying data types.
- It supplies proofs for the required properties of these data types.
- It defines the automata with greater precision.
- It provides more details for the proofs of invariance and simulation (although, in some cases, it suppresses routine algebraic reasoning that the Larch Prover carries out automatically).

We discovered no serious errors in the invariance and simulation proofs, but we did discover some missing steps and ambiguities. One missing step involved a simple invariant, $pc_i = snap \Rightarrow op_i = label \vee op_i = scan$, that follows "by inspection," but was neither stated nor proved in the manual proof. Another involved the well-definedness of the transition relation. For an I/O automaton to be well-defined, whenever an action is enabled in a reachable pre-state, there must be a post-state satisfying the transition relation. Although this fact is fairly "obvious" for UCTS, it is by no means obvious for BCTS and depends on a careful handling of the technicality mentioned at the end of Section 2. Gawlick does address this technicality, but less formally than we do.

The computer-assisted proof involves extensive case analysis, as well as explicit commands to eliminate existential quantifiers in axioms and conjectures. Figures 11 and 12 illustrate the similarities and differences between the manual and computer-assisted proofs. Figure 12 contains the portion of our proof corresponding to the typical proof fragment in Figure 11. The two proofs have roughly the same size and structure, but the organization of the computer-assisted proof is more transparent. For example, it contains separate lemmas to show $b'.t_{max} = b'.t_k$ and $b'.i_{max} = k$. In the computer-assisted proofs, kc, bc, and $b'c$ are constants corresponding to k, b, and b' in the manual proof; the *fix* command applies the definition of *tmax* in state $b'c$ to produce a process index ic; the *critical-pairs*

```
resume by case kc ≠ imax(bc) ∧ tmax(bc) < bc[kc].nt
  prove tmax(b'c) = b'c[kc].t
    resume by case b'c[kc].t < tmax(b'c)
      fix i as ic in descendants($tmaxDef) / c-o(b'c)
      resume by case ic = kc
        critical-pairs *caseHyp with *hyp, $tmaxDef
        resume by case ic = imax(bc)
          instantiate l1 by tmax(bc), l2 by bc[kc].nt in TimeStamp
          instantiate i by ic, j by imax(bc), k by kc in descendants($inv1Def)
          resume by case tmax(bc) = tmax(b'c)
            critical-pairs *caseHyp with $tmaxDef
            instantiate l1 by tmax(bc), l2 by bc[kc].nt in TimeStamp
            instantiate l1 by tmax(b'c), l2 by bc[kc].nt in TimeStamp
            critical-pairs *caseHyp with $tmaxDef
      instantiate l1 by tmax(b'c), l2 by b'c[kc].t in TimeStamp
      critical-pairs *caseHyp with $tmaxDef
      instantiate i by kc in $tmaxDef
  prove kc = imax(b'c)
    resume by cases imax(b'c) < kc, kc < imax(b'c), kc = imax(b'c)
      instantiate j by kc in proper-descendants($imaxDef)
      instantiate y by kc, x by imax(b'c) in TotalOrder
      instantiate i by imax(b'c) in $tmaxDef
      resume by case (bc[imax(b'c)]).t < tmax(bc)
        resume by case imax(b'c) = imax(bc)
          resume by case kc = imax(b'c)
            instantiate
              i by imax(b'c), j by imax(bc), k by kc in descendants($inv1Def)
              ..
            instantiate j by imax(b'c) in *Hyp
        resume by case kc = imax(b'c)
          instantiate j by imax(b'c) in *Hyp
          instantiate i by imax(b'c) in $tmaxDef
```

Figure 12 Computer-assisted proof of Claim 6.4.6

and *instantiate* commands replace variables by appropriate constants (such as *ic*) in definitions and case hypotheses; the Larch Prover then uses these facts to prove the stated conjectures by rewriting them to *true*. Note that the computer-assisted proof does not need to appeal to a high-level notion of a "choice vector." The proofs of these two lemmas are typical and require no human interaction other than what is shown.

An MIT freshman produced the first version of our proof in only 8 weeks, learning about distributed algorithms and the Larch Prover in the process. Checking his proof required about an hour of CPU time on a 100mHz DEC Alpha. We spent a good deal of subsequent time polishing the proof because we believe that such polishing can improve all proofs, whether manual or computer-assisted, by clarifying the presentation, highlighting important aspects, and suppressing less important aspects. We polished the computer-assisted proof by:

- Making it correspond more closely to Gawlick's manual proof, after it had diverged somewhat because of various technical details that arose in using the Larch Prover.
- Decreasing the length of the proof by finding systematic ways of handling similar cases.
- Raising the level of abstraction in the guidance supplied to the Larch Prover, thereby making it easier to modify the proof and recheck it after each modification.
- Decreasing the amount of time required to recheck the proof (by memoizing guidance that the Larch Prover had discovered automatically).
- Making the correctness of the formulation more apparent, as discussed below.
- Reducing the number of axioms. This both made the soundness of the axioms more apparent and improved the performance.

Since one of the primary goals of computer assistance is to increase confidence in the correctness of a proof, we paid particular attention to ensuring that our proof did not simply replace one source of uncertainty (e.g., gaps in the manual proof) by another (e.g., questions about whether the computer-assisted proof established the same result as the manual proof, or whether some subtle inconsistency had crept into the axiomatization). In order to keep our formalization, as shown in Figures 2-10, readable by the distributed algorithms community, we chose not to use a prover like the Boyer-Moore (1988) prover, which forces its users to code axioms and conjectures in a style that precludes inconsistencies at the expense of readability. Instead, we analyzed our axiomatization for potential sources of inconsistency, and we proved lemmas to show that various definitions did not introduce inadvertent inconsistencies.

Explicit (i.e., nonrecursive) definitions and definitions by induction over freely generated types are guaranteed to be consistent. Thus our axioms for I/O automata, natural number indices, and finite sequences are all consistent (at least relative to Peano arithmetic, which justifies the consistency of recursive definitions). We proved lemmas to establish the consistency of other definitions, e.g., implicit definitions and recursive definitions over nonfreely generated types. For example, to show that the implicit definition of *tmax* in UCTS does not introduce an inconsistency, we proved the following easy lemma:
$\exists l(\exists i(s[i].t = l) \wedge \forall i(s[i].t \leq l))$
In order to show the same for the corresponding definition of *tmax* in BCTS, we proved a similar, but nontrivial lemma, using ideas sketched by Gawlick:
$\forall i \forall j \forall k\, trans(s[i].t, s[j].t, s[k].t) \Rightarrow \exists l(\exists i(s[i].t = l) \wedge \forall i(s[i].t \leq l))$
In this manner, we showed that the definitions of *imax*, *tmax*, *full*, and *nextlabel* in Figures 3, 5, and 6 do not introduce any inconsistencies.

We exercised particular care with special purpose traits such as Symbol, TimeStamp, BCTS, and UCTS. Since Symbol axiomatizes a finite type, we verified its consistency by checking its axioms in a finite model; in addition, we used the Knuth-Bendix completion procedure in the Larch Prover to show that its equational axiomatization contains no inconsistencies. For general purpose traits (such as those axiomatizing arrays, ordering relations, and finite sets), we relied on short, published specifications (Guttag and Horning, 1991). We were careful to combine these specifications only in ways that did not doubly constrain relations like $<$.

Table 1 shows the number, structure, and size of our specifications and computer-assisted proofs. We specified axioms for all relevant data types by Larch traits, and we formalized all relevant lemmas and theorems as implications of those traits. For each trait, Table 1 lists those other traits that contribute axioms to it, the number of new axioms

Table 1 Larch traits used in proof

Trait	Subtraits, included (and implied)	Numbers of axioms	Numbers of lemmas	Numbers of proof lines
Array		2	0	0
BCTS	CTS, TimeStamp, IndexSet;	20		
	$\Rightarrow$ ExistsPostState, InductiveInv		12	≈ 1500
BoundedMap	Finite, TotalOrder	0	1	10
CTS	Array, IOAutomaton, ZeroToN	31	0	0
ExistsPostState	IOAutomaton	1	0	0
Finite	Sequence	1	0	0
FiniteSet		7	0	0
IOAutomaton		17	0	0
IndexSet	ZeroToN, FiniteSet	2	5	70
InductiveInv	IOAutomaton	2	0	0
PartialOrder		3	4	15
Permutation	Finite	5	0	0
RationalOrder	TotalOrder	3	0	0
Sequence		3	0	0
SimulationBU	BCTS, UCTS; $\Rightarrow$ SimulationMap	1	0	≈ 600
SimulationMap	IOAutomaton	2	0	0
Symbol		14	12	10
TimeStamp	ZeroToN, Symbol, Array	9	15	125
TotalOrder	PartialOrder	1	2	5
UCTS	CTS, RationalOrder;	5		
	$\Rightarrow$ ExistsPostState, InductiveInv		3	30
ZeroToN	TotalOrder, Finite	8	0	0

it introduces, those traits it implies (with proper instantiations for their parameters), the number of lemmas and theorems we proved about it (in addition to the axioms of its implied traits), and a rough count of the number of lines of guidance we supplied to the Larch Prover to prove these lemmas. These numbers are necessarily somewhat arbitrary, because any finite number of axioms can be conjoined into a single axiom, and because the total number of lines of guidance can be reduced by combining several lines into one. However, the statistics were not artificially manipulated, and they reflect accurately where effort had to be invested: in proving that BCTS preserves its invariants, in proving that F is indeed a forward simulation relation, and in formalizing and proving elementary properties of the symbols used for timestamps.

5 ASSESSMENT

The task of transforming Gawlick's 20-page proof into a machine-checked version was not unduly difficult. This was mainly because the proof was already in a style, based on invariants and a forward simulation, that matched the capabilities of our theorem-proving tools. We have found this style to be very successful for manual proofs of many other complicated distributed algorithms, so there is every reason to believe that machine proofs are feasible for these algorithms as well.

There are many benefits that accrued from our proof effort:

Clarity. Contrary to some expectations, the attention we needed to pay to details in carrying out our machine proof did not lead to clutter in the final proof. Rather, the computer-assisted proof process yielded clarity by removing ambiguities, enforcing precise notation, and suppressing routine detail (e.g., straightforward algebraic manipulations).

The structure of the resulting proof is more apparent than that of the manual proof. Working out the proofs in complete detail helped us discover additional structure and lemmas, as well as to identify case splits and applications of axioms and lemmas. The Larch Prover handled many trivial cases automatically. In nontrivial cases, it expanded and simplified high-level abstractions used in the manual proof, thereby making it easier to distinguish the important parts of the abstractions from the technicalities. All this makes the computer-assisted proof easier to read than the manual proof.

Reliability. Our computer-assisted proof supplies details for all cases, thereby eliminating a major source of oversights and errors in ordinary proofs, where similar cases are often dismissed with short phrases like "by analogy" or "similarly." Such reasoning is particularly error prone in the case of concurrent algorithms, where nondeterministic execution makes analogies harder to define and more dangerous to use. The computer-assisted proof eliminates lacunae at the cost of additional work that should, in fact, be in every good manual proof.

Easier rechecking. No matter how carefully one formulates axioms and conjectures before carrying out a large proof, it is likely that invariants, simulation relations, or other definitions will have to be modified during the course of carrying out the proof. Furthermore, even after the proof is complete, one may wish to polish or revise it. These activities can require rechecking large portions of an already-completed proof.

Such rechecking is much safer and easier with a computer-assisted proof than a manual proof. Computer-assisted rechecking eliminates the chance of overlooking a case or assuming properties that are no longer true after a change: the user can rely on the Larch Prover to identify all the places where the changes have an effect. Furthermore, if one uses appropriately abstract proof strategies, rechecking a proof is easy and fast.

Experimentation. Computer-assistance makes it easier to experiment with a proof. One can try to restructure a proof to make it more concise, to "flow" more intuitively, to use better high-level abstractions, or to isolate key lemmas. In all cases, the Larch Prover can be used to determine the impact of the modifications.

Furthermore, the Larch Prover can provide statistics about how many times and where an axiom was used. In this way, axioms that are never used can be identified and removed, and those that are seldom used can be reconsidered. In this way, users can often generalize their results.

Reusable structure. If a computer-assisted proof (or manual proof) is sufficiently modular, its pieces can generally be reused directly for related proofs. In addition, because of the ease of rechecking, a computer-assisted proof can be used as a template for other similar proofs.

On the other hand, there are some drawbacks to computer-assisted proof efforts such as ours. A computer-assisted proof requires all assumptions and claims to be stated precisely;

while that is generally an advantage, it can also be a disadvantage during early states of algorithm and proof development, where it may be easier to think intuitively and imprecisely. However, it is not necessary to *prove* all assertions in early stages of proof development: the Larch Prover allows the user to assert lemmas without having to prove them immediately. A computer-assisted proof also requires formalizing all data types. Although this can be tedious, it is not a recurring cost: each data type only needs to be defined once, and useful lemmas can be accumulated as they are proved.

In our current framework, an annoying requirement is that all components of the post-state must be specified, rather than only those that change. Also, the user must specify proof strategies that are often more detailed than we would wish (e.g., to deal with properties such as transitivity).

There are still potential sources of error in computer-assisted proofs. It is always possible to make mistakes in formalizing or transcribing an algorithm. Using natural, readable notations helps with this problem, as does having many computer tools (e.g., a prover, a simulator, and a model-checker) use the same representation. As discussed in Section 4, inconsistent axiomatizations can be another source of error. However, as discussed there, we feel that proving appropriate lemmas is the proper way to address consistency, rather than using more restrictive logical formalisms, which may increase the likelihood of formalization errors. Furthermore, an automated prover can formulate lemmas whose proof will ensure consistency.

6 CONCLUSIONS AND FUTURE WORK

We have provided a machine-checked proof for the Bounded Concurrent Timestamp algorithm of Dolev and Shavit, based on a pre-existing manual proof by Gawlick, Lynch, and Shavit. This demonstrates the feasibility of machine verification of full-scale distributed algorithms (with arbitrary numbers of processes), provided that a good manual proof is already available. Although our proof was carried out using the Larch Prover, we expect that much of the work should be portable to other theorem provers.

We believe that the effort we put into *formalizing* the algorithm was completely repaid by the help it provided in clarifying ambiguities. In fact, we believe that the technology works so well that designers of similarly complicated distributed algorithms should routinely formalize their algorithms with the same degree of precision and computer assistance. On the other hand, we are not yet prepared to advocate that all such algorithms should be verified mechanically; the work required to do this is still quite substantial and time-consuming. Still, even if non-experts cannot yet easily produce computer-assisted proofs, they can read them and use them to polish their manual proofs. Furthermore, when the algorithm's correctness is especially critical (e.g., when it is to be used for a safety-critical application or as a building block for other algorithms), we believe that the extra work is justified.

We are currently working to stylize and facilitate the proof process by defining a formal language for expressing I/O automata. This language will provide a clear discipline for describing I/O automata, and it will make the programming task easier in other ways (e.g., by providing a default treatment of unchanged state components, which now need to be described explicitly, and by supporting the description of systems as compositions of automata). We are also working on tools to check the syntax and static semantics of automata descriptions, to generate lemmas that must be proved to establish the consistency of these descriptions, to translate these descriptions into input for theorem provers, simulators, and model checkers, and to generate guidance (e.g., proof strategies) for the operation of these tools. We believe that progress on this combination of languages and tools will put computer-assisted proofs within the reach of the entire distributed algorithms community.

REFERENCES

Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. (1993) Atomic snapshots of shared memory. *Journal of the ACM*, **40(4))**, 873-890.

Robert S. Boyer and J Strother Moore. (1988) *A Computational Logic Handbook*, Academic Press.

Danny Dolev and Nir Shavit. (1989) Bounded concurrent timestamps are constructible. *SIAM Journal of Computing*, to appear. Also in *21st ACM Symposium on Theory of Computing*, 454–465.

Cynthia Dwork and O. Waarts. (1992) Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible! *ACM Symposium on Theory of Computing.*

Stephen Garland and John Guttag. (1991) A guide to LP, the Larch Prover. TR-82, DEC Systems Research Center. Updated version available as `http://larch.lcs.mit.edu:8001/larch/LP/overview.html`.

Rainer Gawlick. (1992) Concurrent timestamping made simple. Master's Thesis, MIT EECS.

Rainer Gawlick, Nancy Lynch, and Nir Shavit. (1992) Concurrent timestamping made simple. *Israel Symposium on Theory and Practice of Computing.*

John Guttag and James Horning. (1993) *Larch: Languages and Tools for Formal Specification.* Springer-Verlag.

A. Israeli and M. Li. (1987) Bounded time stamps. *28th Annual Symposium on Foundations of Computer Science*, 371–382.

Leslie Lamport. (1974) A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, **78(8)**, 453–455.

Leslie Lamport. (1986). On interprocess communication. Parts I and II. *Distributed Computing*, **1**, 77–101.

Victor Luchangco. (1994) Using simulation techniques to prove timing properties. Master's Thesis, MIT EECS.

Victor Luchangco, Ekrem Söylemez, Stephen Garland, and Nancy Lynch. (1994) Verifying timing properties of concurrent algorithms. *FORTE'94.*

Nancy Lynch. (1996) *Distributed Algorithms.* Morgan Kaufmann.

Nancy Lynch and Mark Tuttle. (1987) Hierarchical correctness proofs for distributed algorithms. *Technical Report MIT/LCS/TR-387*, MIT Laboratory for Computer Science.

Nancy Lynch and Frits Vaandrager. (1991) Forward and backward simulations — Part I: Untimed Systems. (1995) *Information and Computation*, **121(2)**, 214–233.

Anna Pogosyants. (1994) Incorporating specialized theories in a general purpose theorem prover. Master's Thesis, MIT EECS.

Nir Shavit. (1989) Concurrent timestamping. Ph.D. Thesis, The Hebrew University.

Jørgen Sogaard-Andersen, Stephen Garland, John Guttag, Nancy Lynch, and Anna Pogosyants. (1993) Computer-assisted simulation proofs. *4th Conference on Computer Aided Verification.*

P. Vitanyi and B. Awerbuch. Shared register access by asynchronous hardware. *27th Symposium on Foundations of Computer Science.*