A Mechanically Checked Proof of a Multiprocessor Result via a Uniprocessor View

J Strother Moore[*]

August 22, 1998

Abstract

We describe a mechanically checked correctness proof for a system of n processes, each running a simple, non-blocking counter algorithm. We prove that if the system runs longer than $5n$ steps, the counter is increased. The theorem is formalized in applicative Common Lisp and proved with the ACL2 theorem prover. The value of this paper lies not so much in the trivial algorithm addressed as in the method used to prove it correct. The method allows one to reason accurately about the behavior of a concurrent, multiprocess system by reasoning about the sequential computation carried out by a selected process, against a memory that is changed externally. Indeed, we prove general lemmas that allow shifting between the multiprocess and uniprocess views. We prove a safety property using a multiprocess view, project the property to a uniprocess view, and then prove a global progress property via a local, sequential computation argument. Our uniprocessor view is a formal compositional semantics for a shared memory system.

1 Informal Discussion of the Problem

Consider a system of n processes each executing the five step program in Figure 1 against a shared memory. The execution model we use is that each of the five instructions is atomic and they are executed in an interleaved way by the various processes. Naively, each process is simply incrementing a shared global counter, `CTR`, non-atomically. The variables `old` and `new` are local to each process.

Instruction 2 is just a "compare and swap" (`CAS`). The instruction either writes `new` to `CTR` or reads `CTR` into `old`. In either case, it sets `new` to a Boolean indicating which branch was taken. The type of `new` is thus integer or Boolean. The program could be simplified by deleting instruction 3 and using instruction

[*]Department of Computer Sciences, University of Texas, Austin, TX 78712, moore@cs.utexas.edu.

```
0. old := CTR;
1. new := old+1;
2. if CTR = old
     then CTR:=new; new:=false;
     else old:=CTR; new:=true; endif
3. if new then goto 1; else skip; endif
4. goto 0;
```

Figure 1: The Algorithm

4 to loop back to the top in both cases. As written, instruction 4 serves as a placeholder for whatever computation is done after `CTR` is incremented by this process.

We assume that the local state of each process initially satisfies certain properties, such as that the program counter points to one of the instructions above and that `new` is equal to `old`+1 whenever the program counter points to instruction 2. A system state satisfying these properties is said to be a "good state." It is easy to prove that good states exist and that the property is invariant under system execution.

It is also easy to prove the safety property that the value of the counter weakly increases over time. We call this the "weak monotonicity" property. Only the "then-clause" of instruction 2 changes the counter. Because the then-clause is executed only when the counter's value is `old`, and because it sets the counter to `new`, which is `old`+1 by our good state invariant, instruction 2 either increments the counter or leaves it unchanged. Hence, the weak monotonicity property holds.

More interesting is a progress property:

> **Informal Main Theorem**.
> Given enough time, the counter will strictly increase.
> **Informal Proof**:
> Consider some process in the system. There is some number, k, such that no matter what the initial program counter, k sequential steps of the process will have caused the process to execute instruction 2. When instruction 2 is commenced, either the value of the counter is that of `old` or it is not. If the former, then the instruction increments the counter. If the latter, then the counter must have changed since we loaded `old`. But by weak monotonicity, it can only have increased.

This informal argument is interesting for several reasons. First, it is flawed but suggestive of a convincing argument. Second, it takes a "single process" perspective on a problem inherently concerned with concurrency. While intuitively appealing because it permits us to reason by symbolic execution, this perspective is confusing in two ways. It leaves implicit the idea that the counter is being changed by other processes during the k steps considered. Furthermore, the "it can only have increased" is at best a very brief summary of the reconciliation between the single process view and the global view. A third reason the argument is interesting is that it leaves implicit an appeal to the pigeon hole principle: some process executes at least k steps if the system runs long enough.

In essence, the purpose of this paper is to describe a formalization of this argument in ACL2 [4, 9]. In so doing we will also demonstrate that the "intuitively appealing" argument can be made entirely precise and rigorous. Furthermore, our proof makes clear the validity and utility of shifting between the concurrent, multiprocess view and the sequential, uniprocess view as appropriate.

2 Related Work

The non-blocking counter algorithm discussed here is trivial and well-known in the community concerned with non-blocking algorithms. The idea of using non-atomic operations to implement atomic ones is apparently first discussed in Lamport's early papers, e.g., [11]. See [8] for a discussion of wait-free synchronization and the power of various synchronization primitives such as atomic reads and write, compare and swap, etc.

This paper is not about non-blocking algorithms *per se* but rather about a style of proof one might use to deal with certain kinds of concurrent algorithms. We can categorize proofs into those conducted in formal logics — logics in which the syntax, axioms and rules of inference are made explicit and rigidly respected in proofs — and those conducted in the informal (but often very precise) style of traditional mathematics. Most proofs in the literature are in the traditional style, e.g., [8], although many such proofs about non-blocking algorithms are sufficiently subtle that their authors have found it necessary to adopt very precise notation, e.g., [14].

Our proof, while informally presented here, is actually carried out in the formal — indeed the mechanized — logic of ACL2. ACL2, which stands for "A Computational Logic for Applicative Common Lisp," is a general-purpose first-order essentially quantifier-free logic of total recursive functions based on an applicative subset of Common Lisp [18]. ACL2 is a re-implemented and extended version of the Boyer-Moore theorem prover, Nqthm [2]. Its primary use has been the formal specification and verification of microprocessor designs. Readers interested in some background on ACL2 should see the Appendix of this paper.

Two widely-used formal logics designed explicitly for reasoning about concurrent systems are Unity [5] and TLA [12]. In [6] Goldschlag formalizes Unity in the Nqthm logic and then uses Nqthm to derive many Unity proof rules and to check Unity proofs. However, with the exception of [6], and some ongoing work at several labs, Unity and TLA do not currently have much mechanical proof support. A distinguishing characteristic of our proof of the non-blocking counter algorithm is that it was mechanically checked.

Proofs of the correctness of the non-blocking counter algorithm can easily be constructed by hand in these other two formal systems. Indeed, Rajeev Joshi, a University of Texas graduate student working with Jay Misra, has constructed a Unity proof of the counter algorithm in response to the work reported here [personal communication]. Joshi proves both the safety and progress properties. We discuss Joshi's proof in the conclusion of this paper.

3 Formalization

We formalize the problem so that we can speak clearly about the relationship between the multiprocessor view and the uniprocessor view of the system. We formalize the problem in Common Lisp, which is supported by the ACL2 theorem prover. To save space we do not show the formal definitions of all the concepts involved. Nor do we state every lemma in our script leading ACL2 to our proof. Our point here is not to focus on ACL2 but on a formalization of an intuitive style of reasoning about concurrency. We give informal readings of most of our formulas, to clarify concepts not defined here. All the formal definitions and theorems are available at http://www.cs.utexas.edu/users/moore/publications/index.html.

A *multiprocess state* or *m-state* is a triple consisting of the process association list ("alist") mapping processes to their local states as described below, a memory alist mapping variable identifiers to values, and the code listing the instructions each process is executing.

If p is associated with some local state in the process alist of m-state s, we say that p "is a process of" s.

The *local state* of a process is a pair containing a program counter (a number indicating which instruction in the code of the system's m-state is the next to be executed) and a register alist mapping local variable identifiers to values.

We are interested only in m-states containing the following code. The instructions below correspond to the lines in the pseudo-code of Figure 1.

```
'((LOAD OLD CTR)         ;0
  (INCR NEW OLD)         ;1
  (CAS CTR OLD NEW)      ;2
  (BR NEW 1)             ;3
  (JUMP 0))              ;4
```

Each instruction is represented as an object (namely a list containing symbols indicating the opcode and operands). We formalize the semantics of this programming language operationally. We define a "single step" function `mstep` which takes a process, p, and an m-state, s, and steps the state by appropriately "modifying" the local state of process p (rebinding p to a new local state in the process alist of s) and possibly the memory component of s. The `mstep` function returns s if the given p is not a process of s. Following Lisp syntax we write `(mstep p s)`, rather than `mstep`(p, s), to denote the application of the function `mstep` to p and s.

We then define

```
(defun mrun (s oracle)
  (if (endp oracle)
      s
    (mrun (mstep (car oracle) s)
          (cdr oracle))))
```

which takes a list of processes, *oracle*, and an m-state, s, and returns the final m-state obtained by successively stepping s on each p in *oracle*. Thus, the oracle specifies how long the system is run and the particular order in which the processes execute.

Let `(CTR s)` be `(binding 'CTR (mem s))`. That is, let `CTR` be the function that maps an m-state s to the value of the memory location named `'CTR` in the memory of s. The main theorem we will prove is

Main Theorem.
```
(implies (and (good-statep s)
              (every-element-a-processp L s)
              (< (* 5 (cardinality L)) (len L)))
         (< (CTR s) (CTR (mrun s L))))
```

which says that if s is a good state, every element of the oracle L is a process of s, and the length of L exceeds five times the number of distinct processes in L, then the value of `CTR` is strictly increased by running s with oracle L.

The predicate `(good-statep s)` is true iff s is an m-state in which every local process state satisfies a certain "local state invariant" given below, and, in addition, `(CTR s)` is an integer and `(code s)` is the program shown above. The local state invariant is that the program counter of the local state is an integer between 0 and 4, the local `old` contains an integer value *old*, and the local `new` contains $1 + old$, whenever the local program counter is 2.

It is straightforward to show that good states are preserved by `mstep` and hence, inductively, by `mrun`. Formally, the theorem for `mstep` is

```
(implies (good-statep s)
         (good-statep (mstep p s))).
```

This theorem is proved by expanding the definition of `mstep` and considering the cases, i.e., proving that `good-statep` is preserved by every instruction executed by any process. The theorem for `mrun` is

```
(implies (good-statep s)
         (good-statep (mrun s L)))
```

and is trivial to prove by induction on L using the aforementioned **mstep** result.

The weak monotonicity (or safety) property mentioned in the introduction can be formalized for **mstep** as

```
(implies (good-statep s)
         (<= (CTR s) (CTR (mstep p s)))).
```

This is easy to prove by expanding the definition of **mstep**.[1] The only instruction that changes **CTR** is the **CAS** and it either increments it by one (given the local state invariant for p) or leaves it unchanged. A simple induction proves the **mrun** version of weak monotonicity.

4 The Single Process View

To carry out the progress proof sketched, we formalize the single process view of the system. In this view, a state consists of a local process state, the memory, and the code. We call such a state a "uniprocess state" or "u-state." Since a u-state contains the local state of a single process, the notion of running a u-state must provide for the possibility that the shared memory is changed "concurrently" by the processes not represented in the u-state. We formalize this with another "oracle," M, that specifies the sequence of memories seen by successive steps of the single process.

```
(defun urun (us M)
  (cond ((endp M) us)
        ((endp (cdr M))
         (make-u-state (uls us)
                       (car M)
                       (ucode us)))
        (t (urun (ustep us (car M)) (cdr M)))))
```

The functions **uls** and **ucode** take a u-state and return the local process state and code components, respectively. The function **make-u-state** constructs a new u-state from the three components. The function **ustep** takes a u-state and a memory and steps the local process, using the specified memory. Observe that the process is stepped using successive memories in the oracle M. When M contains only one memory, the run terminates by inserting that memory into the final u-state.

There is no *a priori* relation between the successive memories seen by the given process, since we make no assumptions about what the other processes are doing. However, the weak monotonicity theorem will allow us to constrain the value of the counter in successive memories of the oracle when running the counter program.

[1] We could state a stronger safety property with the same high-level proof: if s is a good state then `(CTR (mstep s))` is either `(CTR s)` or `(CTR s)`+1.

5 Relating the Two Views

Before we consider the particulars of the counter program, we prove some general results relating the multiprocessor view, `mrun`, to the uniprocessor view, `urun`. These results will allow us to shift perspective.

For every m-state computation (with oracle L and initial m-state s) there are many "corresponding" u-state computations obtained by taking the viewpoint of one of the processes and using an appropriately constructed memory oracle, M.

Consider the idea of transforming an m-state s to a u-state for a selected process, p. This "projection" of s simply creates the u-state with the same memory and code as s but with the local state of p in s. We denote the projection by `(proj` *p s*`)`. It is easy to construct the appropriate M from L and s: Partition L into the regions between successive occurrences of the selected p. Use `mrun` on the regions to generate the memory to be used by the subsequent p step, starting with the initial state s. Finally, collect the successive memories. This projection of the oracle is formally written `(proj-oracle` *p s L*`)`. Observe that if there are n occurrences of p in L then the projected oracle has length $n + 1$.

One easy consequence of the foregoing definitions is the

Commutative Diagram Theorem.
```
(implies (processp p s)
         (equal (urun (proj p s)
                      (proj-oracle p s L))
                (proj p (mrun s L)))).
```

This is a general result relating the two views of the system and is independent of the code being executed. Informally, the theorem says that if p is a process, then the uniprocessor model, running on the p projections of s and L, produces the p projection of the multiprocessor model, running on s and L.

As noted above, the oracle projection function employs `mrun`, the multiprocessor model, to determine the memory oracle for use by the uniprocessor. The projection function appears on the "uniprocessor side" of the equation above. Thus, there is a sense in which the multiprocessor model is still involved in the uniprocessor run addressed by the theorem. But that use of the multiprocessor model is *compositional* in the uniprocessor model: the behavior of a sequence of instructions in process p can be analyzed with the uniprocessor model without having simultaneously to consider the individual actions of instructions in other processes. Our uniprocessor model is a formal compositional semantics for a shared memory system. This proof of **Crux** below illustrates this.

Since the projection function, `proj`, preserves the memory of the projected m-state s, we can derive the following theorem by taking the `mem` of both sides of the above equality and simplifying:

Commutative Diagram Corollary.

```
(implies (processp p s)
         (equal (umem
                  (urun (proj p s)
                        (proj-oracle p s L)))
                (mem (mrun s L)))).
```

That is, we can reason about the final memory of an m-state run by reasoning about the final memory of a u-state run from any p projection, provided that p is one of the processes of the m-state. Since our main theorem concerns the final memory value of `CTR` in an m-state run, the theorem above is the formal tool that allows us to take the single process view.

A second consequence of the definitions of `proj` and `proj-oracle` is a way to project results such as our weak monotonicity theorem for m-states into analogous theorems about u-states. Recall that the weak monotonicity result is that the value of `CTR` weakly increases in an `mrun` from a good state. Then, roughly speaking, the value of `CTR` weakly increases in the memories produced by `proj-oracle` from a good state.

Rather than prove this theorem about our particular program and its treatment of `CTR`, we again prove a more general result — a result independent of a particular program. Suppose we have a property ψ of m-states that is preserved by `mstep`, and suppose we have a preorder (a reflexive and transitive relation) $\mathcal{R}$ on memories with the property that when ψ holds on s, $\mathcal{R}$ holds between the memory of s and the memory of `(mstep p s)`. Then $\mathcal{R}$ holds between the successive memories seen by `ustep` starting from `(proj p s)` with memory oracle `(proj-oracle p s L)`.

Preorder Projection Theorem.

```
(implies (and (ψ s)
              (processp p s))
         (all-R (proj p s)
                (proj-oracle p s L)))
```

where `all-`$\mathcal{R}$ checks that $\mathcal{R}$ holds between successive `ustep`s as noted above.

The **Preorder Projection Theorem** holds for any ψ and $\mathcal{R}$ with the properties noted above. In ACL2 we state this "second order" theorem by using ACL2's "encapsulation" mechanism [10] to constrain new functions ψ and $\mathcal{R}$ to have the properties noted. Then we define `all-`$\mathcal{R}$ in terms of $\mathcal{R}$ and `ustep`. **Preorder Projection** is proved by an induction on L, decomposing it by regions between successive occurrences of p as in `proj-oracle`.

We then "functionally instantiate" [3] the theorem to obtain

Preorder Projection Corollary.

```
(implies (and (good-statep s)
              (processp p s))
         (ascendingp (proj p s)
                     (proj-oracle p s L)))
```

where

```
(defun ascendingp (us M)
  (cond ((endp M) t)
        (t (and (integerp (binding 'CTR (car M)))
                (<= (binding 'CTR (umem us))
                    (binding 'CTR (car M)))
                (ascendingp (ustep us (car M))
                            (cdr M))))))
```

To derive this functional instance we let ψ be `good-statep` and let $\mathcal{R}$ be the relation defined by `(`$\mathcal{R}$ *mem*1 *mem*2`)` = `(and (integerp (binding 'CTR` *mem*2`)) (<= (binding 'CTR` *mem*1`) (binding 'CTR` *mem*2`)))`. It is easy to show that these choices satisfy the constraints on ψ and $\mathcal{R}$, namely that the former is preserved by `mstep` and the latter is a preorder for good state memories and that successive `mstep`s are related by it. The function `ascendingp` is just `all-`$\mathcal{R}$ under this instantiation of $\mathcal{R}$.

In essence, the **Preorder Projection Theorem** allows us to "trade in" weak monotonicity at the m-state level for the assurance that the memories seen at the u-state level satisfy the `ascendingp` predicate. Observe that with the exception of the invariance of `good-statep` and weak monotonicity, we have not yet had to reason about a particular program: we are proving general results about two different ways of viewing a concurrent system.

6 The Crux of the Proof

It is now time to address ourselves to the program in question. Thanks to the **Commutative Diagram Corollary** we can consider a uniprocess view under a projection of the multiprocess oracle. Thanks to the **Preorder Projection Theorem** we know the memories in the projected oracle satisfy the `ascendingp` property. We will use these results formally in the next section to derive our main theorem. In this section we simply prove a uniprocessor result under an assumption of `ascendingp` memories.

Recall the flawed informal argument on page 2. The argument focussed on a number of local steps, k, sufficient to insure that the process executes the `CAS` instruction. Analysis of the code reveals that any path of length $k = 5$ will execute the `CAS`. But the first arrival at the `CAS` is not sufficient to insure that the counter increases from the multiprocess perspective. In particular, suppose that the process arrives at the `CAS` while `old` is less than the current value of the counter. Then the `CAS` will not change the counter. The informal argument proves that the counter will have increased between the time the selected process loaded `old` and the time it executes the `CAS`, but not that it increased during the k instructions.

To carry off a proof from the uniprocess perspective, it is sufficient to argue that during any uniprocess run of a fixed length k, the selected process both reads the counter and subsequently executes the `CAS`. The code reads the counter in two places, instruction 0 and instruction 2. Analysis of the code reveals that every path of length 6 reads the counter and subsequently executes the `CAS`. The most interesting such path is the one that starts at the `BR` at program counter 3, with an outdated version of `old` and a true `new`. This initial path condition obtains if the process had executed the `CAS` (obtaining a good `old` and a true `new`) and had then been suspended long enough for other processes to increment the counter. Proceeding from that initial path condition the process (1) jumps to instruction 1, (2) sets `new` to 1+`old`, (3) executes the `CAS` to set `old` appropriately (but not change the counter), (4) executes the `BR`, again, to jump to 1, (5) sets `new` appropriately and, finally, (6) executes the `CAS` that either increments the counter or demonstrates that it has increased during this run.

Formally, the theorem, below, says that if M is of length 7 and is ascending from a good u-state us,[2] then the value of `CTR` strictly increases in a `urun` from us with M. The fact that M must be of length 7 is a "fencepost" phenomenon: such an M causes the uniprocess to step six times because `urun`, above, terminates without stepping when M has one element.

Crux.
```
(implies (and (equal (len M) 7)
              (ascendingp us M)
              (good-u-statep us))
         (< (binding 'CTR (umem us))
            (binding 'CTR (umem (urun us M)))))
```

This theorem is proved automatically by case analysis on the program counter of the local state followed by expansion of the operational semantics of the subsequent six instructions. Such proofs are often said to be "by symbolic execution." Such a finite case analysis and symbolic execution cannot be carried out without first abstracting away from the arbitrary number of "irrelevant" steps carried out by the other processes between steps of the selected one. This is why the uniprocess view (or compositional semantics) is useful.

7 The Main Theorem

The main theorem says that `CTR` goes up if we run the multiprocessor system on any oracle longer than a certain number. We first prove a lemma that stitches together all of the foregoing results.

Suppose we have a good m-state s and an oracle L in which some process p of s occurs exactly six times. Then `(CTR` s`)` is strictly less than `(CTR (mrun` s L`))`.

[2]Here `(good-u-statep` us`)` checks that the local state of us enjoys the local state invariant and the code of us is our program.

Lemma.

```
(implies (and (good-statep s)
              (processp p s)
              (equal 6 (occurrences p L)))
         (< (CTR s)
            (CTR (mrun s L))))
```

Proof: Instantiate **Crux** above, replacing *us* by `(proj p s)` and replacing M by `(proj-oracle p s L)`. The result is:

Crux'.

```
(implies (and (equal (len (proj-oracle p s L)) 7)
              (ascendingp (proj p s) (proj-oracle p s L))
              (good-u-statep (proj p s)))
         (< (binding 'CTR (umem (proj p s)))
            (binding 'CTR (umem (urun (proj p s)
                                      (proj-oracle p s L))))))
```

By the **Commutative Diagram Corollary** and the definitions of `CTR` and `proj`, the conclusion of **Crux'** is `(< (CTR s) (CTR (mrun s L)))`, which is the conclusion we wish to prove. Therefore, it remains only to show that the hypotheses of **Lemma** imply those of **Crux'**. The first hypothesis is easy: if p occurs six times in L then the length of `(proj-oracle p s L)` is seven. The second we get from the **Preorder Projection Corollary**: if s is a good state and p is a process of s, then `(proj-oracle p s L)` is ascending from `(proj p s)`. The third follows trivially. **Q.E.D.**

The diagram in Figure 2 illustrates the proof. The oracle L has a distinguished process p. The m-states shown are labelled s and `(mrun s L)`. They map, respectively, to the u-states u and `(urun s M)`. The oracle L maps to M as `(proj-oracle p s L)`. By the **Preorder Projection Corollary**, M is ascending. We wish to prove that $ctr < ctr'$, where ctr is the value of `CTR` in the memory of s and ctr' is the value in `(mrun s L)`. The value of `CTR` in u is also ctr. By the **Commutative Diagram Corollary**, ctr' is the value of `CTR` in `(urun s M)`. But by **Crux**, $ctr < ctr'$.

The **Lemma** requires that the selected p occur exactly six times in the oracle. What if it occurs more times? Then the counter still strictly increases because the first six occurrences give us a strict increase and subsequent occurrences do not reduce the counter (by weak monotonicity). So we have

Lemma'.

```
(implies (and (good-statep s)
              (processp p s)
              (<= 6 (occurrences p L)))
         (< (CTR s)
            (CTR (mrun s L))))
```

We may therefore address ourselves to the problem of finding, in sufficiently long oracles, a p that occurs six or more times.

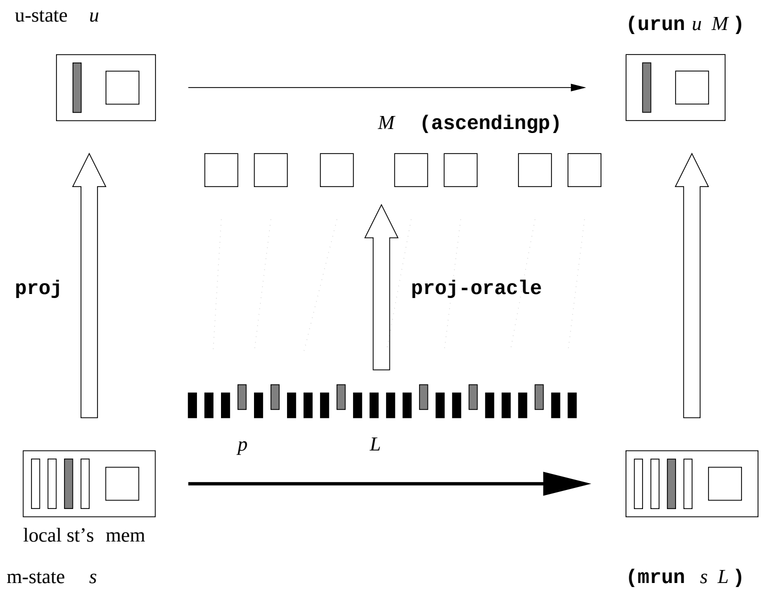

Figure 2: Diagram of Proof of Lemma

But if the length of L exceeds i times the number of distinct processes in L, then there exists a p that occurs more than i times. This is just a version of the

Pigeon Hole Principle.

```
(implies (and (integerp n)
              (<= 0 i)
              (< (* i (cardinality L)) (len L)))
         (< i (occurrences (choose i L) L)))
```

where `(choose i L)` is recursively defined to find an element of L that occurs more than i times or return `nil`.

Furthermore, if every element of L is a process of s then `choose` will return a process of s. Thus, by instantiating **Lemma'**, letting p be `(choose 5 L)` and appealing to the **Pigeon Hole Principle** to establish the third hypothesis of **Lemma'**, we get:

Main Theorem.

```
(implies (and (good-statep s)
              (every-element-a-processp L s)
              (< (* 5 (cardinality L)) (len L)))
         (< (CTR s) (CTR (mrun s L)))).
```

8 Conclusion

It is illuminating to contrast our proof with the previously mentioned work by Joshi. Joshi's proof of the safety property in Unity is virtually identical to our ACL2 proof of that property, modulo the differences between the two logics. Roughly speaking, both proofs merely consider every possible step by every process and show in every case that the counter cannot decrease.

Joshi's proof of our main theorem appears fundamentally different from ours. In Unity, our main theorem is stated as a "leads to" property. It is proved by Joshi using a well-foundedness argument. He defines a measure on (what we would call) m-states and a well-founded relation on that measure. The measure is essentially the lexicographic combination of two measures, the number of processes for which `old=CTR` and a vector of size n containing the program counters of the n processors (normalized so that the `CAS` instruction has the lowest program counter). Joshi then proves that any step by any process decreases this measure according to the obvious lexicographic ordering.[3] Thus, the main work in Joshi's progress proof is at the multiprocessor level: the system is viewed globally and the measure is responsible for providing a coherent view of the effects of any step by any process.

In contrast, the program-specific work in our proof, namely the proof of **Crux**, is carried out at the uniprocessor level. We reason about the sequential execution of a single, selected process. The process we select is not necessarily the one responsible for ultimately incrementing the counter. Indeed, our proof does not identify the responsible process.

We can reason locally because our **Commutative Diagram Corollary** gives us a way to project the system's global behavior into the view from any selected process. The effects of the other processes are captured in the safety property and then transformed, via functional instantiation to our **Preorder Projection Corollary**, which provides us with a local view of the global memory from the perspective of the selected process.

Whether our approach can be adapted to more interesting concurrent systems remains to be seen. This is our first experiment. If this is a promising approach it can probably be used in other formal systems. Just as the ACL2 and Unity proofs of the safety property are essentially the same, one can well imagine carrying out a proof of this nature with Unity after the derivation of an appropriate derived rule of inference for Unity. However, it is important that the formal logic employed have the expressive power to define such concepts as our **`ascendingp`**.

We have illustrated a method of proving a theorem about a system of concurrent programs. The method allows one to reason accurately about the behavior of a multiprocess system by reasoning about the sequential computation carried

[3]Joshi actually proves a stronger theorem than ours. He omits our "good state" initial condition. To do this, he has to elaborate his measure with a third component, which records how many processes have read `CTR`.

out by a selected process against a memory that is changed externally. We find this method appealing primarily because it corresponds closely to the "naive" way of reasoning about such programs. That is, the naive approach, if formalized carefully, provides a rigorous verification methodology. Indeed, it can be mechanically checked by a pre-existing tool.

9 Acknowledgments

I thank Bobby Blumofe for bringing the non-blocking counter algorithm to my attention. I also thank Rajeev Joshi and Pete Manolios, who provided valuable feedback on an early draft of this paper.

10 Appendix: Background on ACL2

ACL2 stands for "A Computational Logic for Applicative Common Lisp." ACL2 is both a mathematical logic and system of mechanical tools which can be used to construct proofs in the logic. The logic formalizes a subset of Common Lisp. The ACL2 system is essentially a re-implemented extension, for applicative Common Lisp, of the so-called "Boyer-Moore theorem prover" Nqthm [1, 2].

10.1 The Logic

The ACL2 logic is a first-order, essentially quantifier-free logic of total recursive functions providing mathematical induction and two extension principles: one for recursive definition and one for "encapsulation."

The syntax of ACL2 is a subset of that of Common Lisp. Formally, an ACL2 term is either a variable symbol, a quoted constant, or the application of an n-ary function symbol or lambda expression, f, to n terms, written `(`$f\ t_1 \ldots t_n$`)`. This syntax is extended by Common Lisp's facility for defining constant symbols and macros.

The rules of inference are those of Nqthm, namely propositional calculus with equality together with instantiation and mathematical induction on the ordinals up to $\epsilon_0 = \omega^{\omega^{\omega^{\cdots}}}$.

The axioms of ACL2 describe five primitive data types: the complex rationals, characters, strings, symbols, and ordered pairs or lists. The complex rationals are complex numbers with rational components and hence include the rationals, the integers and the naturals. Symbols are logical constants denoting words, such as `CAS` and `mstep`. Symbols are in "packages" which provide a convenient way to have disjoint name spaces. `SMITH::mstep` is a different symbol than `JONES::mstep`; but if the user has selected `"SMITH"` as the "current package" then the former symbol can be written more succinctly as `mstep`.

Essentially all of the Common Lisp functions on the above data types are axiomatized or defined as functions or macros in ACL2. By "Common Lisp functions" here we mean the programs specified in [18] that are (i) applicative, (ii) not dependent on state, implicit parameters, or data types other than those in ACL2, and (iii) completely specified, unambiguously, in a host-independent manner. Approximately 170 such functions are axiomatized or defined.

Common Lisp functions are partial; they are not defined for all possible inputs. In ACL2 we complete the domains of the Common Lisp functions and provide a "guard mechanism" by which one can establish that the completion process does not affect the value of a given expression. See [9].

Finally, ACL2 has two extension principles: definition and encapsulation. Both preserve the consistency of the extended logic. The definitional principle insures consistency by requiring a proof that each defined function terminates. This is done, as in Nqthm, by the identification of some ordinal measure of the formals that decreases in recursion.

The *encapsulation* principle allows the introduction of new function symbols constrained by axioms to have certain properties. Consistency is preserved by requiring the exhibition of *witness* functions proved to have the alleged properties. One may then define functions and prove theorems about the constrained symbols. The *functional instantiation* mechanism, essentially the same as described in [3] for Nqthm, then allows the derivation of new theorems from old ones by the apparently higher order act of replacing constrained and defined function symbols by other function symbols — with the obligation of proving that the new symbols satisfy the constraints on the old.

10.2 The System

Like Nqthm, ACL2's theorem prover orchestrates a variety of proof techniques. As suggested by Figure 3, the user puts the formula to be proved into a pool. The simplifier, the most important proof technique, draws a formula from the pool and either "simplifies" it — replacing it in the pool by the several new formulas sufficient to prove it — or passes it to the next proof technique. The simplifier employs many different proof techniques, including conditional (back chaining) rewrite rules, congruence-based rewriting, efficient ground term evaluation, forward chaining, type-inference, the OBDD propositional decision procedure, a rational linear arithmetic decision procedure, and user-defined, machine-verified meta-theoretic simplifiers.

Roughly speaking, as the formula moves clockwise around the ring in Figure 3 it becomes more general. Eventually, if all else fails, the induction mechanism is applied.

The proof techniques are extensions of those used by Nqthm; see [1]. Most of the techniques are rule-driven. The rules are derived from previously proved theorems. For example, if the user has instructed ACL2 to prove that `append` is associative

```
(equal (append (append x y) z)
```

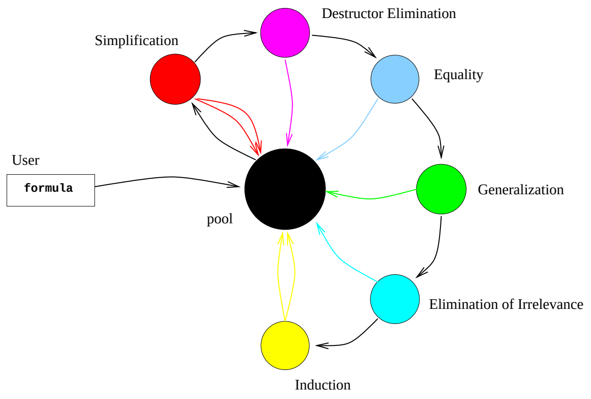

Figure 3: The Orchestration of Proof Techniques

```
    (append x (append y z)))
```

and to use that fact as a **:rewrite** rule (from left to right), then — after the associative law is proved — the simplifier will right-associate all **append**-nests.

ACL2, like Nqthm, “overloads” formulas so that in addition to their usual logical interpretation the system interprets them as rules. The behavior of the system is determined by the rules (theorems) in its database. The data base may be configured by the user by including various certified “books” of previously proved rules. New rules can be added only by causing ACL2 to prove the corresponding theorem. See Figure 4. The user is therefore responsible for codifying a proof strategy as a set of theorems. The system is responsible for soundness.

10.3 Applications

Among the significant proof projects carried out with ACL2 are

- the verification of the floating-point division and square root microcode for the AMD K5^{TM}[13, 15],
- the verification of the IEEE compliance of the RTL for the AMD K7^{TM} floating-point addition, subtraction, multiplication, division and square root [16],
- the ACL2 modeling of the Motorola CAP digital signal processor and its use to prove that a pipeline hazard detection predicate was correct and that several DSP microcode applications were correct [4],

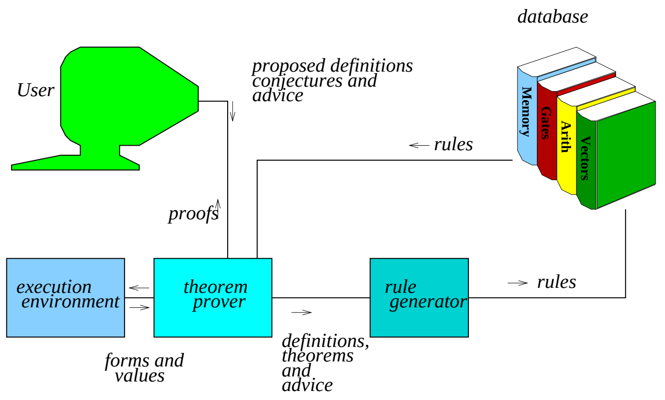

Figure 4: System Architecture

- the modeling of a pipelined microprocessor with multiple, out-of-order instruction issue with a reorder buffer, speculative execution and exceptions and proofs that relate relate this model to a more conventional ISA model [17],
- the modeling of the ALU of the JEM1 microprocessor, the world's first silicon Java Virtual Machine [7].

ACL2 distributed without fee under the GNU General Public License. The ACL2 home page, http://www.cs.utexas.edu/users/moore/acl2, contains an annotated bibliography, a user's manual and instructions for how to obtain the system.

References

[1] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press: New York, 1979.

[2] R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition*, Academic Press: London, 1997.

[3] R. S. Boyer, D. Goldschlag, M. Kaufmann, and J S. Moore. Functional Instantiation in First Order Logic. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Academic Press, 1991, pp. 7-26.

[4] B. Brock, M. Kaufmann and J S. Moore, "ACL2 Theorems about Commercial Microprocessors," in M. Srivas and A. Camilleri (eds.)

Proceedings of Formal Methods in Computer-Aided Design (FM-CAD'96), Springer-Verlag, pp. 275–293, 1996.

[5] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison Wesley: Massachusetts, 1988.

[6] D. M. Goldschlag, Mechanizing Unity, in M. Broy and C. B. Jones (eds.), *Programming Concepts and Methods*, North Holland: Amsterdam, 1990.

[7] D. Hardin, M. Wilding, and D. Hardin, "Transforming the Theorem Prover into a Digital Design Tool: From Concept Car to Off-Road Vehicle," in A. J. Hu and M. Y. Vardi (eds.) *Computer Aided Verification: 10th International Conference, CAV '98*, Springer-Verlag LNCS 1427, 1998.

[8] M. Herlihy, Wait-Free Synchronization, *ACM Trans. Prog. Lang. and Sys.* **11**(1), pp. 124–149, January, 1991

[9] M. Kaufmann and J Strother Moore "An Industrial Strength Theorem Prover for a Logic Based on Common Lisp," *IEEE Transactions on Software Engineering*, **23**(4), pp. 203–213, April, 1997

[10] M. Kaufmann and J Strother Moore, "Structured Theory Development for a Mechanized Logic," URL http://www.cs.utexas.edu/-users/moore/acl2/reports/km98.ps.

[11] L. Lamport, Concurrent reading and writing, *Commun. ACM*, **20**(11), pp 806-811, November, 1977.

[12] L. Lamport, The Temporal Logic of Actions, TOPLAS, **16**(3) pp. 872–923, May, 1994.

[13] J S. Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5$_K$86 Floating Point Division Algorithm, *IEEE Trans. Comp.*, **47**(9), (to appear) September, 1998.

[14] M. Moir, Fast, Long-Lived Renaming Improved and Simplified, *Science of Computer Programming* (to appear). URL: http://www.cs.-pitt.edu/~moir/Papers/moir-scp97.ps.

[15] D. Russinoff, "A Mechanically Checked Proof of Correctness of the AMD5$_K$86 Floating-Point Square Root Microcode," *Formal Methods in System Design Special Issue on Arithmetic Circuits*, 1997.

[16] D. M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of the Floating Point Multiplication, Division, and Square Root Algorithms of the AMD-K7^{TM} Processor URL `http://www.onr.com/-user/russ/david/k7-div-sqrt.html`.

[17] J. Sawada, W. Hunt, Jr., Processor Verification with Precise Exceptions and Speculative Execution, *Computer Aided Verification 1998*, Lecture Notes in Computer Science, Springer Verlag, 1998.

[18] G. L. Steele, Jr. *Common Lisp The Language, Second Edition.* Digital Press, 30 North Avenue, Burlington, MA 01803, 1990.