Internal Note #216
Computational Logic, Inc.
February 26, 1991

A simple example for nqthm: modeling locking

Matt Kaufmann

February 12, 1991

Yesterday I visited with Stan Letovsky of Andersen Consulting, Chicago. The purpose of the visit was to explore the possibility of using Boyer-Moore theorem prover [Boyer-Moore 88, Boyer-Moore 79], or something like it, in some capacity related to their part in the KBSA (Knowledge-Based Software Assistant) project, [kbsa 83]. Stan proposed a little example for us to work on, which I'll describe presently.

ACKNOWLEDGEMENT: I'd like to thank Stan Letovsky for suggesting this example and for taking the lead in formalizing the problem for the Boyer-Moore theorem prover.

Consider a transaction-based system in which transactions each have identification numbers and there are READ, WRITE, and possibly other kinds of transactions. Let's say that a READ and a WRITE transaction are ''corresponding'' exactly when they have the same identification number. In this example, we want to have some sort of ''locking'', in the sense that once a READ transaction occurs, the corresponding WRITE transaction must occur before another READ occurs.

Stan and I modeled such an implementation yesterday and began a proof of a formalization of this correctness property. Today I modified and finished this effort, and the result is the list of events shown below. An important data structure here is that of an ''object'', as recognized by the shell recognizer function OBJ?, which has two fields. The first field is the LOCK-TID field, which is either F (false), meaning that there is no pending WRITE needed to correspond to the last READ that took place, or is the transaction i.d. of the last READ transaction. The second field is the WAITERS field, which is a queue of transactions that have been set aside while waiting for an appropriate WRITE to take place.

Thus, the implementation function SERVER takes a ''transaction queue'' TRANQ and an ''object'' X and returns a ''trace'' of the transactions that took place. The idea is that once a READ has occurred in TRANQ, everything else is queued up in the WAITERS field of X until the corresponding WRITE occurs in TRANQ, at which time the WAITERS queue is appended to the front of what is left of TRANQ. Also at that time, the system is ''unlocked'' in the sense that the LOCK-TID field of X is set to F (false).

Two important notions for the statement of our correctness property are those of a ''good transaction queue'', or GOOD-TRANQ, and of a ''good transaction trace'', or GOOD-TRACE. A good transaction queue is a list of transactions with the property that every READ is followed by a corresponding WRITE (exactly one WRITE corresponding to each READ). See the definition of GOOD-TRANQ below for a more precise definition. A good transaction trace is a list of transactions with the property that for each READ transaction in the list, no other READ transaction appears in the list until the corresponding WRITE

transaction appears. See the definition of GOOD-TRACE below for a more precise definition. Finally, the correctness theorem (the final theorem below, SERVER-SAFETY) states that if TRANQ is a good transaction queue, then the result of running the SERVER on TRANQ is a good transaction trace.

The proof was developed originally with an integers-and-vectors library loaded, but afterwards I eliminated that in favor of one definition and two lemmas (as explained in comments below). This event list replays in unadorned nqthm (I actually used a CLI core image where functional variables were loaded in as well), extended by the "nqthm-patches" file from Version 1.3 of pc-nqthm [Kaufmann88-1 88]. That file was loaded in because it contains an extension to the nqthm syntax that allows the familiar COND, CASE, and LET constructs. The total replay time (real time) on a Sun 3/60 was about 67 seconds.

```
;; The following rule is essentially in the library I loaded.
(prove-lemma append-associativity (rewrite)
  (equal (append (append x y) z)
         (append x (append y z))))

;; Essentially from the library I loaded.
(defn length (lst)
  (if (listp lst)
      (add1 (length (cdr lst)))
    0))

(add-shell make-op nil op?
           ((op-type (none-of) zero)
            (tid (one-of numberp) zero)))

(add-shell make-obj nil obj?
           ((lock-tid (one-of numberp falsep) zero)
            (waiters (none-of) zero)))

;; oops -- another basic lemma needed
(prove-lemma length-append (rewrite)
  (equal (length (append x y))
         (plus (length x) (length y))))

(defn server (tranq x)
  (let ((current (car tranq)))
    (cond ((nlistp tranq)
           ())
          ((and (equal (op-type current) 'write)
                (equal (tid current) (lock-tid x)))
           (cons current
                 (server (append (waiters x) (cdr tranq))
                         (make-obj f nil))))
          ((falsep (lock-tid x))
           (if (equal (op-type current) 'read)
               (cons current (server (cdr tranq)
                                     (make-obj (tid current) (waiters x))))
             (cons current (server (cdr tranq) x))))
          (t (server (cdr tranq)
                     (make-obj (lock-tid x)
                               (append (waiters x) (list current))))))))
  ((ord-lessp (cons (add1 (plus (length (waiters x)) (length tranq)))
                    (length tranq)))))

(defn find-write (tid tranq)
```

```
  (if (listp tranq)
      (or (and (op? (car tranq))
               ;; aha!  Forgot the following -- see explanation below.
               (equal (op-type (car tranq)) 'write)
               (equal tid (tid (car tranq))))
          (find-write tid (cdr tranq)))
    f))

#|
I was trying to prove a lemma (later not needed) which generated
the following goal:

(IMPLIES (AND (LISTP TRANQ)
              (OP? (CAR TRANQ))
              (EQUAL TID (TID (CAR TRANQ))))
         (IMPLIES (FIND-WRITE TID TRANQ)
                  (EQUAL (GOOD-TRANQ (REMOVE-WRITE TID TRANQ))
                         (GOOD-TRANQ TRANQ))))

Inspection of this goal lead me to realize that when removing
or finding a WRITE operation, one needs to check not only the
TID but also that the operation is really a WRITE!  I then fixed
the definition of REMOVE-WRITE (as indicated below), and later
realized I needed to make a similar change to the definition of
FIND-WRITE (see above).
|#

(defn remove-write (tid tranq)
  (if (listp tranq)
      (if (and (op? (car tranq))
               ;; aha!!! I forgot the following.  See comment above.
               (equal (op-type (car tranq)) 'write)
               (equal tid (tid (car tranq))))
          (cdr tranq)
        (cons (car tranq)
              (remove-write tid (cdr tranq))))
    nil))

(prove-lemma good-tranq-helper (rewrite)
  ;; This lemma helps with getting the definition of
  ;; GOOD-TRANQ accepted.
  (not (lessp (length tranq)
              (length (remove-write tid tranq)))))

(defn good-tranq (tranq)
  (if (listp tranq)
      (and (op? (car tranq))
           (if (equal (op-type (car tranq)) 'read)
               (and (find-write (tid (car tranq)) (cdr tranq))
                    (good-tranq (remove-write (tid (car tranq))
                                              (cdr tranq))))
             (good-tranq (cdr tranq))))
    t)
  ((lessp (length tranq))))

(defn good-trace (trace tid)
```

```
  ;; Trace is a list of ops; the tid flag says whether a read is pending
  ;; with the indicated tid (and is f if not).  *** Notice that
  ;; there can be "no-ops", i.e. ops other than 'read and 'write,
  ;; and hence the read and its matching write can be separated.
  (if (listp trace)
      (and (op? (car trace))
           (if tid
               (case (op-type (car trace))
                     (read f)
                     (write (and (equal (tid (car trace)) tid)
                                 (good-trace (cdr trace) f)))
                     (otherwise (good-trace (cdr trace) tid)))
             (case (op-type (car trace))
                   (read (good-trace (cdr trace) (tid (car trace))))
                   ;; *** at one point I had F for the WRITE case,
                   ;; but this was the only case that failed in
                   ;; the main proof once I fixed everything else.
                   ;; Actually, though, if there is a write when
                   ;; there's no pending read, it's a no-op and
                   ;; not any sort of error!  So, I'll comment out
                   ;; the following line:
                   ;; (write f)
                   (otherwise (good-trace (cdr trace) f)))))
    (if tid
        f
      t)))

;; *** The simplification in the case below where tid is F is perhaps
;; the key to the whole proof.  Once I made this change, the only
;; remaining fix was the one in the definition shown above, and that
;; was easy to locate by inspection of the output of the main lemma,
;; SERVER-SAFETY-MAIN-LEMMA.

(defn good-tranq-tid (x y tid)
  ;; This says that (append x y) is a good-tranq except that when tid
  ;; is not F, it says that (append x (remove-write tid y)) is a good-tranq
  ;; and that the appropriate write exists in y.
  ;; Also, I've noticed during the proof that it's an invariant of the system
  ;; that the waiters list is empty when there's no lock on.  So, I add this
  ;; in.

  (if tid
      (and (find-write tid y)
           (good-tranq (append x (remove-write tid y))))
    (and (equal x nil)
         (good-tranq y))))

(prove-lemma server-safety-main-lemma nil
  (implies (and (obj? x)
                (good-tranq-tid (waiters x) tranq (lock-tid x)))
           (good-trace (server tranq x) (lock-tid x))))

(prove-lemma server-safety (rewrite)
             (implies (good-tranq tranq)
                      (good-trace (server tranq (make-obj f nil))
                                  f))
```

```
((use (server-safety-main-lemma
       (x (make-obj f nil))))))
```

References

[Boyer-Moore 79] R. S. Boyer and J S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

[Boyer-Moore 88] R. S. Boyer and J S. Moore.
A Computational Logic Handbook.
Academic Press, Boston, 1988.

[Kaufmann88-1 88]
Matt Kaufmann.
A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover.
Technical Report 19, Computational Logic, Inc., May, 1988.

[kbsa 83] Cordell Green, David Luckham, Robert Balzer, Thomas Cheatham, Charles Rich.
Report on a Knowledge-Based Software Assistant.
Technical Report, Kestrel Institute, 1983.