An Incremental Stuttering Refinement Proof of a Concurrent Program in ACL2

Rob Sumners
Advanced Micro Devices
UT/Austin

ACL2 Workshop
October 31, 2000
[ Overview ]

- Introduction
  - A brief history...
  - Definition of the Concurrent Deque Program
    - The definition and use of records
  - Specification Program

- Stuttering Refinement
  - Definition and Proof Requirements
  - Proof Strategies:
    - Reduction to single-step, Incremental stages, Distribution over process composition, Introduction of auxiliary var.s

- Chain of refinement proofs:
  \[ cdeq \leftrightarrow cdeq+ \leftrightarrow intr \leftrightarrow intr+ \rightarrow spec \]
  - Using the ACL2 proof checker
A Brief History...

- Some time ago, Sandip Ray, Greg Plaxton, and Robert Blumhofe presented their proof of the implementation of a concurrent deque at an ACL2 meeting.

  - The implementation is “wait-free” and was used in a process scheduler based on work-stealing.

- While their statement of correctness was elegant, their proof was complicated by the details of the implementation.

  - It appeared to be a good candidate for ACL2.

- Our approach is to prove that their concurrent program is a stuttering refinement of a much-simpler program whose correctness is (hopefully) apparent.

  - The use of stuttering refinement allows the specification to match any finite number of steps in the implementation with a single step.

  - Consequently, eventual progress in the implementation can be analyzed by examining the possible steps of the specification.
Concurrent Deque Introduction

- The concurrent deque program \texttt{cdeq} consists of:
  - A single \texttt{owner} process which can push values onto and pop values from the bottom of the deque
  - An arbitrary (but fixed) number of \texttt{thief} processes which can pop values from the top of the deque

- Thief processes resolve contention for the top of the deque by testing-and-setting the top pointer of the deque

- The \texttt{Owner} may also contend with the \texttt{Thieves} for the last element in the deque, in which case it may also test-and-set the top pointer
  - In this case, the owner also clears the top and bottom pointers by setting them to memory address 0

- We would like to show that eventually some process pops from a non-empty deque

- \textbf{Convention}: capitalized variables are shared amongst processes, while lowercase variables are local to a process
cdeq state – a record of:

- **shared** – record storing shared var.s:
  - **MEM** – a vector of data values
  - **RET** – the last successful pop
  - **CLK** – labels each pop uniquely
  - **BOT** – **MEM** address of the bottom
  - **AGE** – a pair of numbers:
    - **tag** – uniquely identify same tops
    - **top** – **MEM** address of the top

- **owner** – record storing local var.s:
  - **loc** – current program location
  - **dtm** – next value to push
  - **bot** – local copy of **BOT**
  - **old** – local copy of **AGE**
  - **new** – modification of **old**
  - **itm** – data value to be returned
  - **ret** – a local return value

- **thieves** – a vector of records, where each one stores
  the local var.s of a thief (same as **owner**, w/o **dtm**)

**cdeq** input – a record of:

- **N** – process selector
- **P** – select push or pop
- **D** – data value to push
Step 8 of the thief program and step 14 of the owner program are “compare-and-swap” operations.
Made extensive use of records in the definitions and proofs

- Records are essentially alists where the keys are ordered

- Allows a fixed set of reduction rules for record access and update
  
  o Similar to Matt Wilding and Dave Greve’s rules for \texttt{nth} and \texttt{update-nth}

- Importantly, we can use symbols for the field names which improves the readability of the ACL2 output

- Matt Kaufmann made a significant contribution by removing the \texttt{recordp} hypotheses from the reduction rules

\begin{verbatim}
;; (g a r) -- record get --
;; returns the value stored in field a in record x
;; (s a v r) -- record set --
;; returns a record with the value v stored in field a
;; and all other fields with the values in r

(defthm g-diff-s
  (implies (and (force (fieldp a))
                (force (fieldp b))
                (not (equal a b)))
           (equal (g a (s b v r))
                  (g a r))))
\end{verbatim}
• Definition of the thief next-state program in ACL2

(defun c-thf-s (f s)
  (case (loc f)
    (8 (if (equal (age s) (old f))
      (>s :age (new f))
      (>s :ret (itm f) :clk (1+ (clk s)) s))

    (10 (>s :ret (itm f) :clk (1+ (clk s)))
      (t s))))

(defun c-thf-f (f s)
  (case (loc f)
    (0 (>f :loc 1))
    (1 (>f :loc 2 :old (age s)))
    (2 (>f :loc 3 :bot (bot s)))
    (3 (>f :loc (if (> (bot f) (top (old f))) 5 4)))
    (4 (>f :loc 0 :ret nil))
    (5 (>f :loc 6 :itm (val (g (top (old f)) (mem s))))))
    (6 (>f :loc 7 :new (old f)))
    (7 (>f :loc 8 :new (top+1 (new f)))))
    (8 (>f :loc 9 :new (if (equal (age s) (old f))
      (age s) (new f)))))
    (9 (>f :loc (if (equal (old f) (new f)) 10 11)))
    (10 (>f :loc 0 :ret (itm f)))
    (11 (>f :loc 0 :ret nil))
    (t (>f :loc 0))))
Specification Program, spec

\[
\text{spec}(\text{in})(\text{st})
\]

if in.N then
  if thieves[in.N]
    \( RET \leftarrow \text{thieves}[\text{in.N}] \)
    \( CLK \leftarrow CLK + 1 \)
    thieves[in.N] \leftarrow \text{nil}
  else if steal-last(\text{DEQ}, \text{owner}, \text{in})
    thieves[in.N] \leftarrow \text{owner.itm}
    owner.itm \leftarrow \text{nil}
  else
    thieves[in.N] \leftarrow \text{get-top(\text{DEQ})}
    \text{DEQ} \leftarrow \text{drop-top(\text{DEQ})}
else
  case owner.loc
  \text{PUSH:}
    \text{DEQ} \leftarrow \text{push-bot(owner.dtm, \text{DEQ})}
    owner.loc \leftarrow \text{'IDLE}
  \text{POP:}
    \( RET \leftarrow \text{or(owner.itm, RET)} \)
    CLK \leftarrow CLK + 1
    owner.itm \leftarrow \text{nil}
    owner.loc \leftarrow \text{'IDLE}
  \text{IDLE:}
    \text{if in.push then}
      \text{owner.dtm} \leftarrow \text{in.D}
      owner.loc \leftarrow \text{'PUSH}
    \text{else}
      owner.itm \leftarrow \text{get-bot(\text{DEQ})}
      \text{DEQ} \leftarrow \text{drop-bot(\text{DEQ})}
      owner.loc \leftarrow \text{'POP}

\quad \text{label(\text{st}) = list(CLK, RET, owner.dtm)}
Trace Refinement

- A step function \texttt{impl} is a \textit{trace refinement} (\(\Rightarrow\)) of the step function \texttt{spec} w. r. t. \(\langle\text{label}, \text{inv}\rangle\) if for every \textit{run} of \texttt{impl}, there exists a \textit{run} of \texttt{spec} such that the sequence of labels for each run correlate

  - The predicate \texttt{inv} defines the “well-formed” \texttt{impl} states

- Reasoning about infinite runs is awkward, instead reduce trace refinement to single-step theorems:

\[
\begin{align*}
\text{(defthm labels-equal-=>)} & \\
& \quad \text{(equal (label (rep st)) (label st))}
\end{align*}
\]

\[
\begin{align*}
\text{(defthm inv-persists-=>)} & \\
& \quad \text{(implies (inv st)} \\
& \quad \quad \text{(inv (impl in st)))}
\end{align*}
\]

\[
\begin{align*}
\text{(defthm rep-matches-=>)} & \\
& \quad \text{(implies (inv st)} \\
& \quad \quad \text{(equal (rep (impl in st))} \\
& \quad \quad \quad \text{(equal (spec (pick in st) (rep st)))))
\end{align*}
\]

  - \texttt{rep} maps \texttt{impl} states to \texttt{spec} states and \texttt{pick} chooses an input for a \texttt{spec} state given the current \texttt{impl} state and input

  - Trace refinement requires \texttt{impl} and \texttt{spec} to move in lock-step
Stuttering Refinement

- Alternative is to prove *stuttering refinement* (>>)
  - Trace refinement with “sequence of labels” replaced by “compressed sequence of labels”

- Again, we would like to reduce this to a single-step criterion:

```
(defthm well-founded->>
  (bounded-ordp (rank st) (rank-dep)))
```

```
(defthm rep-matches->>
  (implies (and (inv st)
                 (not (equal (rep (impl in st))
                           (spec (pick in st) (rep st))))
               (and (equal (rep (impl in st))
                       (rep st))
                (e0-ord-< (rank (impl in st))
                          (rank st))))))
```

- Originally defined in [Namjoshi97] and refined in [Manolios99]

- Introduce a **rank** function which maps states to **e0-ordinals** and demonstrate that this measure decreases when the **spec** and **impl** states don’t commute

- A sufficient condition to ensure stuttering equivalence (<>-) is if **pick** is the identity function on **in**
Refinement Proof Strategy

• Stuttering refinement is compositional
  
  \((- (impl \gg intr) \text{ and } (intr \gg spec)) \implies (impl \gg spec)\)
  
  • Allows incremental proof of stuttering refinements by defining intermediate models and then chaining together each intermediate refinement step
  
  • We use intermediate steps to introduce auxiliary variables which help to correlate different step functions
  
  \(cdeq \leftrightarrow cdeq+ \leftrightarrow intr \leftrightarrow intr+ \gg spec\)

• Stuttering refinement distributes over asynchronous process composition

  • If \((\text{spec is sp1||sp2}) \text{ and } (impl is im1||im2) \text{ and } (im1 \gg sp1) \text{ and } (im2 \gg sp2)\) then \((impl \gg spec)\)

  • This property allows us to define the functions \(\text{rep}\) and \(\text{rank}\) component-wise

    ○ For example, \(\text{rep}\) is defined by \(\text{rep-owner}\), \(\text{rep-shared}\), and \(\text{rep-thieves}\). \(\text{rep-thieves}\) is defined as \(\text{rep-thief}\) for each thief process

• Basic goal in defining \(\text{intr}\): component-wise stuttering equivalence
[ Defining intr and (cdeq+ ↔ intr) ]

- An additional goal in defining intr was to translate the deque in MEM to a true-list using:

```lisp
(defun mend (bot top mem)
  (and (integerp bot)
       (integerp top)
       (> bot top)
       (cons (g (1- bot) mem)
             (mend (1- bot) top mem)))))
```

- The strategy in defining intr-thf and intr-onr was to hide local steps:

```
loc    cdeq+-thf(f, S)  loc    intr-thf(f, S)
0      skip             0      ctr ← CTR
1      old ← AGE
       xctr ← XCTR
2      bot ← BOT
       xitm ← and(BOT > AGE.top,
                     MEM[AGE.top])
3      if bot ≤ old.top then 2
        return nil
4      0
5      itm ← MEM[old.top] 2
6      new ← old
7      new.top ← new.top + 1
8      if old = AGE then 2
        new,AGE ← AGE,new
        XCTR ← XCTR + 1
9      if old = new then 0 | 3
10     RETURN itm
11     return nil
12     0 | 3
13     RETURN itm
14     0
```

;; the following test passes iff DEQ

;; was non-empty and we “succeed”
Restructured **rep-matches->>** to afford more direct proof with ACL2

- The predicate **suff** is a sufficient condition for **rep-matches->>**, but is not required to persist
- The predicate **commit** defines the cases when **intr** can match the next **cdeq+** step

```
(defthm >>-stutter1
  (implies (and (suff st in)
                 (not (commit st in)))
            (equal (rep (cdeq+ in st))
                   (rep st))))

(defthm >>-stutter2
  (implies (and (suff st in)
                 (not (commit st in)))
            (e0-ord-< (rank (cdeq+ in st))
                      (rank st))))

(defthm >>-match
  (implies (and (suff st in)
                 (commit st in))
            (equal (rep (cdeq+ in st))
                   (intr (pick in st) (rep st))))))

(defthm >>-invariant-sufficient
  (implies (inv st) (suff st in)))
```
[ Proving (cdeq+ <-> intr) cont’d ]

• After proving some simple rules about the variable translations (see below) the above theorems went through with little or no assistance

(equal (get-top (mend bot top mem)) (val (g top mem)))

• The time required to prove (cdeq+ <-> intr) was essentially the time required to discover the correct definitions and to prove inv-persists->>

  – Several iterations were required to strengthen suff to inv

  ◦ For instance, while the following is sufficient for cdeq+ at loc 8:

    (equal (equal (age s) (old f))
         (= (xctr f) (xctr s)))

  ◦ The invariant required this stronger condition to hold from locs 2-8:

    (if (equal (age s) (old f))
        (= (xctr f) (xctr s))
        (and (age<< (old f) (age s))
             (< (xctr f) (xctr s)))))
Defining and Proving (intr+ >> spec)

- While the nature of (cdeq+ <-> intr) was straightforward, (intr+ >> spec) is a little more subtle
  - Yet, the relative simplicity of intr+ compared with cdeq+ significantly reduced the complexity of proving (intr+ >> spec)

- Since the spec thief does not fail when the deque is non-empty, we need to hide failing intr+ thief executions
  - rank function used in (intr+ >> spec)

(defun rank (st)
  (if (consp (deq (shr st)))
    (cons (cons (rank-onr (onr st))
      (miss-count (tvs st) (max-thf)
        (ctr (shr st))))
      (rank-tvs-non-empty (max-thf) (tvs st)))
    (cons (rank-onr (onr st))
      (rank-tvs-empty (max-thf) (tvs st))))

- Once the proper definitions were discovered, the proof of (intr+ >> spec) was essentially automatic
  - The added non-determinism in spec allows us to hide the detail of when a thief can steal at the cost of proving <->
[Using the ACL2 proof checker]

• Finally, I found the ACL2 proof checker to be an indispensible tool for:

  — Working through theorems with large case splits, Analyzing the
type-alist, Diagnosing failed rewrite attempts, Defining pc-macros
for handling repetitive tasks

ACL2 !>(set-inhibit-output-1st '(proof-tree prove))
(PROOF-TREE PROVE)

... additional definitions, theorems ...

... begin interaction cycle ...
ACL2 !>(defun inv-onr (o s) ...)
ACL2 !>(verify (implies (and (inv-shr s)

  (inv-onr o s)
  (assume-thf f s))

  (inv-onr o (c+-thf-s f s)))))
->: bash
***** Now entering the theorem prover *****
... subgoals which failed simplification ...
->: (repeat prove)
... stops on first goal (if any) which fails the full prover ...
... we examine this goal to determine why it failed ...
->: exit
ACL2 !> :u
ACL2 !> (defun inv-onr (o s) ... update the invariant ...)
ACL2 !> (verify (implies (and (inv-shr s) ...
... repeat verify attempt ...
Acknowledgements and Future Work

• Acknowledgements:

  – Ray, Plaxton, and Blumhofe posed the initial challenge

  – Sandip provided additional input and analysis of the work presented

  – Pete made many useful suggestions and pointed out an error in an earlier labeling function

  – Matt made significant improvements to the records book and answered many questions about the proof checker

• Future Work

  – Many concurrent programs seem amenable to this style of verification in ACL2

    o Secure Atomic Transaction Processors, Concurrent Garbage Collectors, ...

  – Currently, we are working on a proof of an implementation of the Bakery algorithm at a micro-architectural level