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 <-> cdeq+ <-> intr <-> intr+ >> 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 `cdeq` consists of:
  - A single `owner` process which can push values onto and pop values from the bottom of the deque
  - An arbitrary (but fixed) number of 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 Owner may also contend with the 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
- Convention: capitalized variables are shared amongst processes, while lowercase variables are local to a process

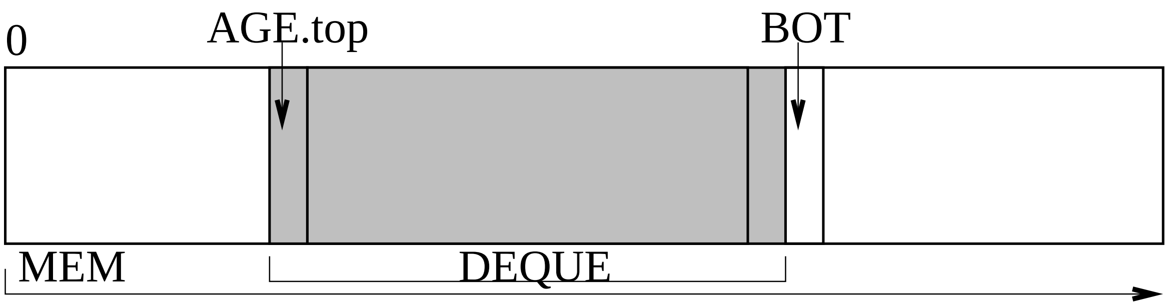
⌊ cdeq state structure ⌋

`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 *top*s
    - *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



⌊ cdeq next-state program ⌋

```
loc   owner(push, D)(o, S)
0     if push then
         dtm ← D
19       bot ← BOT
20       MEM[bot] ← dtm
21       bot ← bot + 1
22       BOT ← bot
      else ;; pop
1        bot ← BOT
2        if bot = 0 then
3           return nil
4        bot ← bot − 1
5        BOT ← bot
6        itm ← MEM[bot]
7        old ← AGE
8        if bot > old.top then
9           RETURN itm
10       BOT ← 0
11       new.tag, new.top ← old.tag, 0
12       new.tag ← new.tag + 1
13       if bot = old.top then
14          if old = AGE then
               new, AGE ← AGE, new
15          if old = new then
16             RETURN itm
17       AGE ← new
18       return nil
```

```
loc   thief()(f, S)
1     old ← AGE
2     bot ← BOT
3     if bot ≤ old.top then
4        return nil
5     itm ← MEM[old.top]
6     new ← old
7     new.top ← new.top + 1
8     if old = AGE then
         new, AGE ← AGE, new
9     if old = new then
10       RETURN itm
11    return nil
```

```
cdeq(in)(st)
if in.N then
   thieves[in.N], shared ←
    thief ()(thieves[in.N], shared)
else
   owner, shared ←
    owner (in.P, in.D)(owner, shared)
```

• Step 8 of the `thief` program and step 14 of the `owner` program are “compare-and-swap” operations

⌊ Defining records in ACL2 ⌋

- 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
    - Similar to Matt Wilding and Dave Greve's rules for **`nth`** and **`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 **`recordp`** hypotheses from the reduction rules

```
;; (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))))
```

⌊ `cdeq` definition in ACL2 ⌋

- Definition of the `thief` next-state program in ACL2

```
(>s :ret (itm f) :clk (1+ (clk s)))
```

macro expands to

```
(s :ret (g :itm f) (s :clk (1+ (g :clk s)) s))
```

```
(defun c-thf-s (f s)
  (case (loc f)
        (8  (if (equal (age s) (old f))
                (>s :age (new f))
              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))))
```

$\lfloor$ Specification Program, `spec` $\rfloor$

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

– $\texttt{label}(st) = \texttt{list}(CLK, RET, owner.dtm)$

⌊ Trace Refinement ⌋

- A step function `impl` is a *trace refinement* (`=>`) of the step function `spec` w. r. t. ⟨`label`, `inv`⟩ if for every *run* of `impl`, there exists a *run* of `spec` such that the sequence of labels for each run correlate

  – The predicate `inv` defines the "well-formed" `impl` states

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

```
(defthm labels-equal-=>
  (equal (label (rep st)) (label st)))

(defthm inv-persists-=>
  (implies (inv st)
           (inv (impl in st))))

(defthm rep-matches-=>
  (implies (inv st)
           (equal (rep (impl in st))
                  (spec (pick in st) (rep st)))))
```

  – `rep` maps `impl` states to `spec` states and `pick` chooses an input for a `spec` state given the current `impl` state and input

  – Trace refinement requires `impl` and `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-depth)))

(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-ordinal`s 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 >> intr`) and (`intr >> spec`)) implies (`impl >> 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 <-> cdeq+ <-> intr <-> intr+ >> spec
  ```

- Stuttering refinement distributes over asynchronous process composition

  – If ((`spec` is `sp1`‖`sp2`) and (`impl` is `im1`‖`im2`) and (`im1 >> sp1`) and (`im2 >> sp2`)) then (`impl >> spec`)

  – This property allows us to define the functions `rep` and `rank` component-wise

    ◦ For example, `rep` is defined by `rep-owner`, `rep-shared`, and `rep-thieves`. `rep-thieves` is defined as `rep-thief` for each thief process

- Basic goal in defining `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:

```
(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	<u>`cdeq+-thf`</u>$()(f, S)$	*loc*	<u>`intr-thf`</u>$()(f, S)$
0	*skip*	*0*	
1	$old \leftarrow AGE$	**0**	$ctr \leftarrow CTR$
	$xctr \leftarrow XCTR$		
2	$bot \leftarrow BOT$	**1**	$itm \leftarrow$ `get-top`(DEQ)
	$xitm \leftarrow and(BOT > AGE.top, MEM[AGE.top])$		
3	if $bot \leq old.top$ then	*2*	
4	return `nil`	*0*	
5	$itm \leftarrow MEM[old.top]$	*2*	;; the following test passes iff DEQ
6	$new \leftarrow old$	*2*	;; was non-empty and we "succeed"
7	$new.top \leftarrow new.top + 1$	*2*	
8	if $old = AGE$ then	**2**	if $and(itm, ctr = CTR)$
	$new, AGE \leftarrow AGE, new$		$DEQ \leftarrow$ `drop-top`(DEQ)
	$XCTR \leftarrow XCTR + 1$		$CTR \leftarrow CTR + 1$
9	if $old = new$ then	$0 \mid 3$	
10	RETURN *itm*	**3**	RETURN *itm*
11	return `nil`	*0*	

⌊ Proving `(cdeq+ <-> intr)` ⌋

- 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 strenghten `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 *loc*s 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 indispensable 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-lst '(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

    - 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