An Incremental Stuttering Refinement Proof of a Concurrent Program in ACL2

Rob Sumners

Computer Engineering Research Center
The University of Texas at Austin
robert.sumners@amd.com

Abstract

We present an incremental refinement proof in ACL2 which demonstrates the reduction of the observable behaviors of a concurrent program to those of a much simpler program. In particular, we document the proof of correctness of a concurrent program which implements the operations of a double-ended queue in the application of a work-stealing algorithm. The demonstration is carried out by proving a refinement from the implementation to a specification via an intermediate model. We document the use of the intermediate model in dividing the verification problem into more manageable steps which in turn allow for more effective proof reductions in ACL2. In both steps, the more abstract system is allowed finite stuttering and this is important in correlating refinement proof with progress in the more concrete system.

1 Introduction

Concurrent program development is an error-prone enterprise. It is difficult for a human programmer to keep track of the various possible states a concurrent program can reach. Thus, an "innocent" change may easily introduce a bug which is hard to detect and/or diagnose since the change was made with an inaccurate mental picture of the program's behavior. It is because of the complexity of bug detection and diagnosis and the ease of bug introduction in concurrent programs, that high-level abstractions are often developed which provide a simpler, safer programming model (e.g. database transactions). In some cases, efficiency is a concern and a low-level concurrent implementation is needed to solve a particular problem. In those cases it is paramount that the programmer carefully documents and/or proves the correctness of his/her algorithm.

Arora, Plaxton, and Blumhofe[2] developed a program for maintaining a deque viewed and manipulated by an arbitrary number of concurrent processes which is used in a process scheduler based on work stealing. The optimality of the scheduler relies on the assumption that the programs manipulating the deque are wait-free but make progress. Plaxton, Blumhofe, and Ray presented a proof of the program's correctness[3] at an ACL2 meeting. Their proof was complicated by the various cases they had to consider in order to cover the possible program states which could be reached. This problem therefore appeared to be an excellent candidate for formalization in ACL2. This paper documents our solution to this problem.

The proof of correctness we present is carried out by showing that the visible behaviors of the concurrent deque implementation correspond to the visible behaviors of a much simpler specification program whose correctness is hopefully apparent. Exhibiting a correspondence between two programs is a common approach to analysis which reflects the principle that it is often easier to specify correctness using a program rather than using formulas.

In this work, our notion of correspondence is *well-founded refinement* which is a reformulation of refinement upto stuttering that is amenable to proof with ACL2. It is derived directly from the work of Namjoshi[8] and Manolios[6, 5] on Well-Founded Equivalence Bisimulations (WEBs). This reformulation significantly decreases the amount of ACL2 proof effort required from the user in proofs involving stuttering. Additionally, the notion of correspondence we use is compositional and thus allows a refinement to be proved in incremental stages. We demonstrate this by breaking the refinement proof for the concurrent deque into four separate refinements which can be chained together to provide the final result.

In Section 3, we will present a precise definition of well-founded refinement and our argument why the allowance of finite stuttering in refinements is appropriate and correct for reasoning about a program at different levels of abstraction. We will then detail in Section 4 the refinement proofs which were carried out including the proof reductions which improved the efficiency of the interaction with ACL2. But first, we present the concurrent deque program which is the target of our formal verification effort.

2 Concurrent Deque

The name "deque" stands for double-ended queue and is a data structure which stores a sequence of elements and supports pushing and popping from both ends of the sequence – or equivalently viewed as a double-ended stack. The program we will analyze is the concurrent deque program presented in Figure 3 and is named `cdeq`. The program `cdeq` is composed of a single owner program and an arbitrary fixed number of thief programs; where each program has access to a common deque. The owner program can push or pop items on the bottom of the deque, while the thief programs can only pop items from the top of the deque. Items are never pushed onto the top of the deque, so the name "deque" is a misnomer. The owner program and each thief program has a local store of variables. The local variables of a program are only accessed and updated by that program. Some additional variables which define the common deque are shared amongst all programs.

It is common in ACL2 modeling of systems or processes with non-terminating behavior to define and reason about the systems as `step` functions. For the purpose of this paper, a `step` function is a binary function which takes a current state and input value and returns the next state. The function `cdeq` in Figure 3 is a `step` function defining the asynchronous composition of the `owner` program with each `thief` program; at each step, either the `owner` takes a step or a `thief` takes a step. The structure of the state and input parameters of `cdeq` are specified in Figure 1. Figure 2 depicts the structure of the deque in memory – the elements of the deque are stored in the indices $[AGE.top, ..., BOT-1]$ where $BOT > AGE.top$. The deque is empty if $AGE.top \geq BOT$ and non-empty otherwise.

In Figures 3 and 5 we use an assignment-style notation for functions which transform states. For each of these functions, the signature is broken into inputs and states. For instance, $\texttt{owner}(push, D)(o, S)$ takes two inputs *push* and D and two states o and S. As a convention, we use uppercase variable names for variables which are shared and lowercase names for variables which are local. For instance, in the body of `owner`, MEM is short for *S.mem* (since S is bound to *shared* in `cdeq`) and *dtm* is short for *o.dtm*. Each of the functions return an updated version of each of its state arguments, where the assignment $field \leftarrow value$ updates the $field$ in the corresponding state with $value$.

The functions `owner` and `thief` define the local step functions. Each local `owner` or `thief` step transforms the state variables depending on the current value of *loc* by performing the corresponding assignments and then updating the *loc* variable to its next value. The program steps were defined [2] to correspond to operations which could be performed atomically for a particular concurrent microarchitecture. For instance, the steps at `owner` *loc* 14 and `thief` *loc* 8 correspond to a common compare-and-swap operation which is often atomic. It should also be noted that the (RETURN *itm*) and (return *nil*) steps in `owner` and `thief` are respectively shorthand for $loc, RET, ret, CLK \leftarrow 0, itm, itm, CLK+1$ and $loc, ret \leftarrow 0, nil$.

As we mentioned before, every thief attempts to pop from the top of the deque. The steps in `cdeq` state – a record of:

- *shared* – a record storing the shared variables:
  - MEM – a natural-addressed vector of data values
  - RET – the last non-`nil` value popped from the deque
  - CLK – an auxiliary label used to tag each pop uniquely
  - BOT – the address of the bottom of the deque in MEM
  - AGE – a pair of numbers:
    - *tag* – used to uniquely identify ages with same *top*
    - *top* – the address of the top of the deque in MEM
- *owner* – a record storing the owner's local variables:
  - *loc* – the current program location
  - *dtm* – stores the value to be pushed onto the deque
  - *bot* – a local copy of the BOT address
  - *old* – a local copy of the AGE address
  - *new* – a modification of *old*
  - *itm* – a local copy of the data value to be returned
  - *ret* – a local return value which is essentially ignored
- *thieves* – a natural-addressed vector of records, where each record stores the local variables of a thief (same as the owner, without *dtm*)

`cdeq` input – a record of:

- N – selects which program (owner or thief) takes the next step
- P – boolean input for owner to select push or pop
- D – data value for owner to push

Figure 1: Structure of `cdeq` state and input parameters

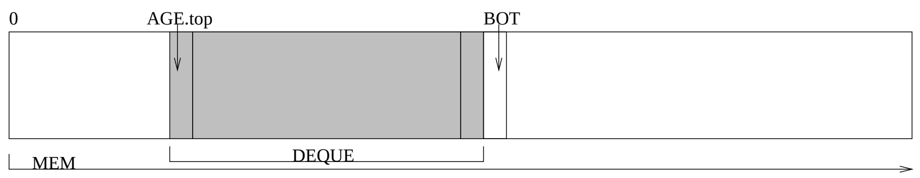

Figure 2: Deque Layout in Memory

```
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)
```

Figure 3: Concurrent Deque Program — `cdeq`

the `thief` function carry out this operation. First, the current AGE and BOT are read into the local variables *old* and *bot*. Next, the thief checks to see if the deque is empty and returns `nil` if it is. Otherwise, the thief will grab the *itm* stored at the top of the deque and then increment a local copy of AGE stored in *new*. At 8, the thief will perform a compare-and-swap which has the result of incrementing $AGE.top$ if the AGE has not changed since it was stored in *old* at 1. The main idea of the `thief` is the point that either a thief "succeeds" in popping the deque – i.e. $AGE = old$ at 8 – or some other process must have changed the AGE of the deque and thereby "succeeded" in their pop. The correctness of the concurrent program only relies on some process succeeding in popping elements from the deque; it doesn't matter which process succeeds. Notice at 9 that *new* is equal to *old* if and only if AGE was equal to *old* at 8 and so the test at 9 and the related returns at 10 and 11 should be clear.

The `owner` function defines steps for implementing a push onto the bottom of the deque, *loc*s 19-22, and for implementing a pop from the bottom of the deque, *loc*s 1-18. Pushes onto the bottom of the deque and pops from the bottom of the deque when the deque has more than one element are independent of pops from the top of the deque and the steps taken in `owner` are straightforward, *loc*s 1-9 and 19-22. When the deque has a single element and the owner wants to pop, then the owner contends with the thieves for that element. This case is handled in *loc*s 10-18. Furthermore, when the owner detects an empty deque, it will reset the deque to $BOT = AGE.top = 0$. When the owner performs this reset of the deque to 0, it increments the $AGE.tag$ to ensure that no stale thief with $old.top = 0$ falsely matches $old = AGE$ at 8 in the `thief` function.

The function `cdeq` is the main step function which defines the asynchronous composition of the `owner` with some arbitrary but fixed number of thieves. A given step is defined by the selection of a process using $in.N$ (either an integer indexing a `thief` or `nil` denoting the `owner`) and then the update of the local state of that process and the shared state using either the `thief` step function or `owner` step function.

We translated the functions in Figure 3 into ACL2 functions. The functions in Figure 4 define the translation of the `thief` function. The function `c-thf-s` returns the updated values for the shared variables for any `thief` step and `c-thf-f` does the same for the local variables. In these functions and many others, we used definitions from a book about records in ACL2. In this context, a record is actually an association list where the entries have been ordered based on a total ordering of the keys. The keys in our case are either symbols or integers and the total ordering on keys is $<$ on integers, `symbol-<` on symbols, and an arbitrary selection of integers ordered before symbols. The records book exports the functions `(g a r)` — get field `a` of record `r` — and `(s a v r)` — set field `a` of record `r` with value `v` — along-with several rewrite rules which simplify terms consisting of record `s`ets and `g`ets. These rewrite rules are provided in appendix A.

The macros `>s` and `>f` used in Figure 4 translate into a sequence of record updates. For instance, `(>f :loc 2 :old (age s))` translates to `(s :loc 2 (s :old (age s) f))`. We also use records for the vectors MEM and *thieves* from Figure 3. For instance, the term `(<- (mem s) (top (old f)))` from Figure 4 translates to `(g (top (old f)) (mem s))`. The use of these record definitions and associated rewrite rules was of paramount importance in improving the readability of proof output from ACL2 (using symbols for keys) while requiring only a few rewrite rules for reducing terms involving record operations. The macros `age`, `new`, `old`, `bot`, `itm`, `mem`, `clk` expand into record accesses of the form `(g :age ..)`, `(g :new ..)`, etc.

Our goal in analyzing the concurrent deque program in Figure 3 is to show that its observable behaviors coincide with the observable behaviors of a much simpler program. Before we consider this simpler program, we first need to define what is observable from any given state and justify this definition. For most programs, the answer to the question "what is observable?" is determined by partitioning the variables into inputs, outputs, and internals, where the output variables are usually considered the observable values[1]. In our case, we want to preserve three variables: the *dtm* variable

[1]For some contexts, it is necessary to preserve the behaviors defined on input variables and output variables, in which case both sets of variables should be considered observable.

```
(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))    ;; we can ignore this no-op step
        (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 (<- (mem s) (top (old f))))))
        (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))))
```

Figure 4: ACL2 definition of Thief function

of the owner state, the *RET* shared variable which is only updated on global RETURNs, and the *CLK* variable which is incremented at every pop in order to distinguish global returns of the same value. The idea is that if we want to verify that the concurrent deque can be viewed ideally as a deque, then we are interested in the values which are pushed onto and popped off of the deque. The observation of a state is defined by a function `label` which takes a state and returns its observable value:

`label`(st) = `list`(*RET*,*CLK*,*owner.dtm*)

We now define a simple program which captures the legal observable behaviors of the concurrent deque. This simple program is given in Figure 5. The thief states have been reduced to a single value which is non-`nil` if the value should be RETURNed or `nil` if the thief should pop the value off the top of the deque. This is exactly what is done in `spec` when *in.N* is non-`nil` except for the case `steal-last` which corresponds to the state where the deque has a single element which the owner has popped but hasn't returned. In this case, a thief can "steal" this last value before the owner can return it. Since we don't care who pops the values off the deque, this is acceptable behavior as long as an element is only popped once. This is ensured by setting *owner.itm* to `nil` which in turn ensures that the owner will not return the element and no other thief can steal the element. The owner state is reduced to a record with three fields *loc*, *itm*, and *dtm*. The variable *owner.loc* goes from 'IDLE to 'POP or 'PUSH and then back to 'IDLE. The functions performed at each of these *loc*s is hopefully clear. When *owner.loc* is POP, it is worth noting that *owner.itm* is RETURNed if and only if it is non-`nil`. As we mentioned before, uppercase variables refer to shared variables and in the function `spec`, the variables *DEQ* and *RET* are really *st.shared.deq* and *st.shared.ret*. The functions `drop-top`, `get-bot`, `push-bot`, `drop-bot`, and `get-bot` are defined on normal ACL2 true-lists as follows:

```
(defun val (x) (or x 0))

(defun first-val (d) (and (consp d) (val (first d))))

(defun get-top (d)
  (if (endp (rest d)) (first-val d)
    (get-top (rest d))))

(defun drop-top (d)
  (if (endp (rest d)) ()
    (cons (first d) (drop-top (rest d)))))

(defun get-bot (d) (first-val d))

(defun drop-bot (d) (rest d))

(defun push-bot (x d) (cons x d))
```

3 Stuttering Refinement

Proofs about functions in ACL2 assume axioms about the ACL2 primitives such as: `(equal (car (cons x y)) x)`. At the ACL2 level we treat functions such as `cons`, `equal`, and `car` as atomic operations which satisfy the axioms we assume. Under the hood, the actual definition for these functions may be several pages of code which requires thousands of machine steps to perform the

```
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

;; NOTE : steal-last(DEQ, -, -) implies DEQ is empty
```

Figure 5: Specification Program — spec

necessary operation[2]. At the ACL2 level, we do not particularly care how `cons`, `car`, and `equal` are implemented under the hood as long as they eventually return values which are consistent with the axioms we assume. Thus, a specification of the implementation for each of the ACL2 primitives in an underlying Lisp environment would consist of (a) each primitive eventually returns a value, and (b) these values are consistent with the axioms of ACL2. Actually, this is invariably incomplete since unlike the nice applicative world of ACL2, low-level machine code has side effects and only works correctly in certain well-formed contexts. For instance, if we executed the code for `cons` in a context where the pointer to the next available cons-cell erroneously pointed to an existing `(cons x y)`, then the setting of the `car` field of this cons-cell may have the side effect of invalidating the axiom `(equal (car (cons x y)) x)`. We therefore adjust the above specification of a Lisp environment by adding that (a) and (b) only have to hold in "well-formed" contexts and then add the condition: (c) the property "well-formed" persists in the execution of each primitive. This concept of a well-formed context or state which persists is often termed an *invariant* and is central to many ACL2 proofs about nonterminating systems. We will use the unary predicate `inv` to define the set of "well-formed" program states.

The various notions of refinement we define are simply formalizations of statements (a), (b), and (c) in the context of nonterminating programs defined by `step` functions. Intuitively, showing that an implementation is a refinement of a specification ensures that the observed behaviors or traces of the implementation are consistent with those of the specification. In order to be more precise, we need to define some terms. A *sequence* X is a total function mapping the natural numbers to ACL2 objects. We will use the notation X_i as shorthand for $X(i)$. For a given unary function `inv`, a *run* R of `inv` is simply a sequence where R_0 is understood as the initial state of the run and is well-formed – $\texttt{inv}(R_0)$ is `T` – and the remaining R_i define an infinite sequence of inputs. For a given binary function `step`, a sequence B is termed a *behavior* of $\langle \texttt{step}, \texttt{inv} \rangle$ if there exists a run R of `inv` such that $B_0 = R_0$, and for all $i > 0$, $B_i = \texttt{step}(R_i, B_{i-1})$. For a given unary function `label`, a sequence T is termed a *trace* of $\langle \texttt{step}, \texttt{label}, \texttt{inv} \rangle$ if there exists a behavior B of $\langle \texttt{step}, \texttt{inv} \rangle$ such that for all i, $T_i = \texttt{label}(B_i)$. A function `impl` is a *trace refinement* of a function `spec` with respect to $\langle \texttt{label}, \texttt{inv} \rangle$ if every trace of $\langle \texttt{impl}, \texttt{label}, \texttt{inv} \rangle$ is also a trace of $\langle \texttt{spec}, \texttt{label}, (\texttt{lambda}(\texttt{x})\texttt{T}) \rangle$. In practice it is difficult to prove (mechanically or otherwise) a trace refinement directly since it involves reasoning about the existence of infinite sequences. A common method of attacking this problem is to define a unary function `rep` – which maps `impl` states to their corresponding or representative `spec` states – and a binary function `pick` – which takes an `impl` state and input and returns a "matching" `spec` input – and proving the following single-step theorems:

```
(defthm labels-equal->>
  ;; we could add assumption of (inv st) here but it's rarely needed.
  (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)))))
```

The theorem `rep-matches->>` is a standard commutative theorem which commonly arises in proofs relating the behaviors of two programs or systems[7]. The combination of `labels-equal->>`

[2] `cons` in Gnu Common Lisp is implemented with the C function make_cons which in the course of allocating a new `cons`, may adjust several counters, allocate a new "page", and/or perform a garbage collection cycle.

and `rep-matches->>` imply that `impl` is a trace refinement of `spec` with respect to $\langle$`label`, `inv`$\rangle$. Unfortunately, with trace refinement we cannot allow the implementation to make moves which are internal or invisible to the specification. Notice that condition (a) does not require that a value is returned immediately, but instead eventually. This distinction is important in order to allow the specification and implementation to be defined at different levels of action granularity. The implementation often needs to be defined with finer-grain actions due to restrictions imposed by the environment in which the implementation is defined. At the same time the specification often needs to be defined with coarser-grain actions for the purpose of clarity and conciseness.[3]

Intuitively, we would like to weaken trace refinement by allowing for finite stuttering and in order to define this, we introduce the notion of sequence compression. The *marker* M of a sequence X is another sequence where $M_0 = 0$ and for all $i \geq 0$, M_{i+1} is defined to be the least integer j such that ($j > M_i$ and $X_j \neq X_{M_i}$) if such an integer exists and $M_i + 1$ otherwise. The *compression* C of a sequence X is then the sequence defined by $C_i = X_{M_i}$. Now we weaken trace refinement to allow for finite stuttering. A function `impl` is a *stuttering refinement* of a function `spec` with respect to $\langle$`label`, `inv`$\rangle$ if every compression of a trace of $\langle$`impl`, `label`, `inv`$\rangle$ is also a compression of a trace of $\langle$`spec`, `label`, `(lambda(x)T)`$\rangle$. Similar to the simplification of trace refinement before, we would like to find a set of single-step theorems which imply a stuttering refinement. This very simplification was defined by Namjoshi[8] and Manolios[6] in the context of bisimulation and for this paper is termed *well-founded refinement* (or simply refinement) and is denoted (`impl >> spec`). The idea is to define the functions `rep` and `pick` as before, but also define a unary function `rank` which maps `impl` states to `e0-ordinals`. Intuitively `rank` defines a well-founded measure which decreases to the next point at which `spec` can match the step of `impl`. The theorems required to prove a well-founded refinement are given in Figure 6. We also added the theorem `well-founded->>` which ensures that the `rank` function returns an `e0-ordinal` (required for well-foundedness w.r.t. `e0-ord-<`) bounded by an ω-stack of depth `(rank-depth)` – i.e. $\underbrace{\omega^{\omega^{\cdot^{\cdot^{\cdot^{\omega}}}}}}_{\texttt{(rank-depth)}}$.

Theorem 1 *If* `impl` *is a well-founded refinement of* `spec` *w.r.t.* $\langle$`label`, `inv`$\rangle$ *then* `impl` *is a stuttering refinement of* `spec` *w.r.t.* $\langle$`label`, `inv`$\rangle$.

As we mentioned before, well-founded refinement is compositional. We used `bounded-ordp` instead of `e0-ordinalp` in `well-founded->>` in order to facilitate the definition of an ordinal pairing of two `rank` functions where `e0-ord-<` on the pair coincides with the lexicographic ordering on the two ranks. In order to prove that we can take two refinements (`impl >> intr`) and (`intr >> spec`) and compose them to conclude (`impl >> spec`), we need the lexicographic ordering since `impl` may stutter in between any stuttering steps of `intr`. Assume we have a refinement (`impl >> intr`) with witness functions `rep1`, `rank1`, `inv1`, and `rank-depth1`; and a refinement (`intr >> spec`) with witness functions `rep2`, `rank2`, `inv2`, and `rank-depth2`. Then we can prove the refinement (`impl >> spec`) with witness functions `rep`, `rank`, `inv`, and `rank-depth` defined by:

```
(defun rank (st)
  (ord-pair (rank2 (rep1 st))
            (rank1 st)
            (rank-depth1)))

(defun rank-depth ()
  (+ 2 (rank-depth1) (rank-depth2)))
```

[3]The distinction here between "coarse-grain" and "fine-grain" is mainly for the purpose of presentation and somewhat arbitrary since you could imagine cases, for instance, where you implement stack operations with array operations and vice-versa.

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

(defthm well-founded->>
  (bounded-ordp (rank st) (rank-depth)))

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

(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)))))
```

Figure 6: Requisite Theorems for Proving Refinement

```
(defun rep (st)
  (rep2 (rep1 st)))

(defun inv (st)
  (and (inv1 st) (inv2 (rep1 st))))

(defun pick (in st)
  (pick2 (pick1 in st) (rep1 st)))
```

As we have stated it to this point, a refinement implies that the compressed traces of `impl` are a subset of the compressed traces of `spec`. But, in some cases we want to show that the compressed traces of `impl` are the same as the compressed traces of `spec`. We term this *strong refinement* and note that a sufficient condition for strong refinement is achieved when the function `pick` is the identity function on its first parameter `in`.[4] Often, strong refinements are preferable since they ensure equivalence of the observed behaviors of the two systems. This is important because an erroneous behavior in the `spec` can be mapped back to an erroneous behavior in the `impl` which may not be the case if `impl` was simply a refinement of `spec`. We will use (`impl <-> spec`) to denote strong refinement.

4 Refinement Proof Details

In this section we outline the definitions and intermediate refinement steps performed in showing that (`cdeq >> spec`) where the relevant functions `cdeq`, `spec`, and `label` were defined in Section 2. We will also detail the steps taken to simplify the proof requirements and some of the ACL2 features which were found to be helpful.

We found the refinement (`cdeq >> spec`) prohibitively complex to prove directly and decided to take the approach of trying to find an intermediate system `intr` which was far simpler than `cdeq`

[4] A more general criterion can be devised for strong refinement, but the simpler requirement that `pick` is the identity function is sufficient for the presentation in this paper.

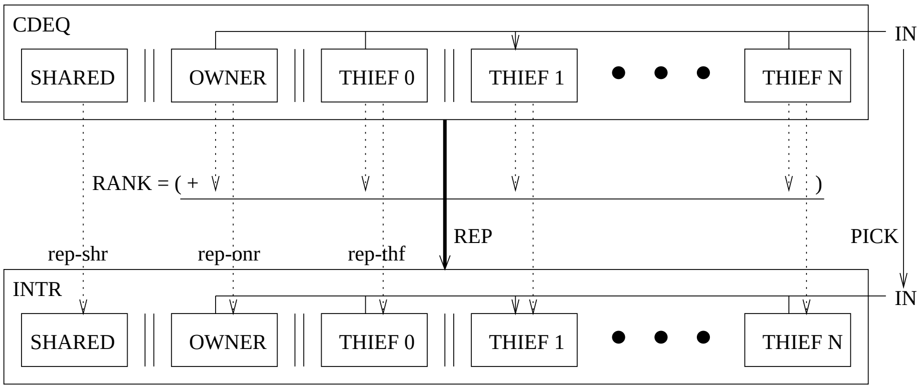

Figure 7: Component-wise definition of `rep` and `rank`

but preserved the observed behaviors of `cdeq`. Thus, our goal was to define a system `intr` such that we could prove (`cdeq <-> intr`). This would allow us to work with `intr` from then on without consideration of `cdeq`.

In analyzing possible goals for defining `intr`, we recognized the need to preserve the compositional structure of `cdeq` in `intr`. This would allow the proof of `rep-matches->>` to be reduced to proving local "matching" theorems for each component; one theorem for the owner and one theorem which could be used for each thief. The local matching theorems would show, in effect, that the each component of `cdeq` was a refinement of the corresponding component of `intr`. The proof of `rep-matches->>` should then follow directly from the local theorems. This proof reduction can be achieved by enforcing the following requirements: (1) `rep` is defined component-wise, (2) `rank` simply adds the ranks of each component, (3) `pick` is the identity function, and (4) `intr` must update the component selected by $in.N$. Now, the proof of `rep-matches->>` simply splits on the component selected by $in.N$ and since the other components (and their local `rep` and `rank` values) remain unchanged, the composite state of `cdeq` stutters iff the selected component stutters and the composite state of `cdeq` is matched by the composite state of `intr` iff the selected component is matched. This reduction lets us define and verify `intr` component-by-component and also is in-line with another goal of proving (`cdeq <-> intr`). In other words, prove (`cdeq-owner <-> intr-owner`) and (`cdeq-thief <-> intr-thief`) and then derive (`cdeq <-> intr`).

Another goal we set for the intermediate model was to translate some of the shared variables in `cdeq` to simpler, coarser definitions in `intr`. First we translated the MEM-based deque in `cdeq` to the list-based deque used in `spec`. This translation is performed by the function `mend` which walks the indices from *bot* to *top* and conses the elements in MEM at each step.

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

Next, we wanted to translate the AGE shared variable to a simple CTR which is incremented any time the thief (and sometimes the owner) pops from the top of the deque. Unfortunately, a consistent value for such a CTR variable cannot be determined by the value of the AGE variable since the owner may set *AGE.top* to 0 at any point. The solution is then to add an "auxiliary" CTR variable to `cdeq` which increments every time the AGE variable is updated. This introduction of auxiliary variables is common and "safe" as long as their values do not affect the actual variables of the program. As mentioned in [1], refinement maps often require the use of auxiliary variables to augment the state of a program with information about its history and its future. Rather than assume the safety of auxiliary variables, we decided to define another system `cdeq+` which consists of `cdeq` along with several auxiliary variables we needed to define the correspondence between `cdeq` and `intr`. We now define `intr` and at the same time augment `cdeq+` as needed and in the end we will prove (`cdeq+ <-> intr`) and (`cdeq+ <-> cdeq`). The proof of (`cdeq+ <-> cdeq`) is straightforward and simply involves the removal of the auxiliary variables in `cdeq+` and requires no stuttering (i.e. `rank` is defined to be 0 and `inv` is defined to be `T`). Thus, `cdeq+` now has an additional shared auxiliary variable $XCTR$ – all auxiliary variables will begin with x or X – which corresponds to the CTR shared variable in `intr`. This completes the translation of the `cdeq` shared variables which is defined by the following function `rep-shr`.

```
(defun rep-shr (sh)
       ;; the macro >_ translates to a sequence of
       ;; updates to an empty record, i.e. ()
  (>_ :deq (mend (bot sh)
                 (top (age sh))
                 (mem sh))
      :ctr (xctr sh)
      :ret (ret sh)
      :clk (clk sh)))
```

The translation of the local owner and thief states follow a similar approach and as such we only document the thief. We want to define the `intr` thief to hide as many steps of the `cdeq+` thief as possible while still preserving the `cdeq+` thief's behavior. Since we cannot hide updates to the shared variables, the goal is then to hide or stutter on "local" steps of the `cdeq+` thief. In Figure 3, *loc*s $3, 4, 6, 7, 9, 11$ are clearly "local" since they do not involve any shared (uppercase) variables. It is also clear that *loc*s 8 and 10 are "global" since a shared variable is updated. This leaves *loc*s $1, 2, 5$ and in order to determine which of these steps can be hidden, we need to consider the behavior of the `cdeq+` thief. At 1, the thief copies AGE to the local *old*. The value in *old* not only determines the item in MEM which may be returned, but more importantly it is used at *loc* 8 to determine if the thief "succeeded". Thus, *loc* 1 clearly cannot be hidden. At *loc* 2, the thief copies the BOT pointer to a local *bot* variable. The value in *bot* is only used at *loc* 3 to determine if the deque is empty. Since the value of BOT may change between *loc*s 1 and 2, we cannot hide the step at *loc* 2. The step at *loc* 5 copies the item in $MEM[\mathit{old.top}]$ to the local variable itm. At first glance it may appear that we cannot hide this step as well. But since we only RETURN itm if $\mathit{old} = AGE$, we know that the top of the deque has not moved and in the `intr` thief, we can grab this value early which in turn allows us to hide the step at *loc* 5. In summary, we have determined for the moment that the `intr` thief only needs to match the steps at *loc*s $1, 2, 8, 10$ in the `cdeq+` thief. Given these steps and the translation of the shared variables described above, we derived the `intr` thief defined in Figure 8. The function `rep-thf` which maps a `cdeq+` thief state to an `intr` thief state is now straightforward: map $xitm$ to itm, $xctr$ to ctr, and `cdeq+` thief *loc* to `intr` thief *loc* as depicted in Figure 8. The function `rank-thf` which defines the local stuttering measure for the `cdeq+` thief is (essentially) $12 - loc$.

The derivation of the definition of the `intr` owner from the `cdeq+` owner followed a similar strategy of hiding "local" steps in the refinement. Referring to the definition of the owner function in Figure 3, the `intr` owner must match the steps at *loc*s $0, 5, 7, 9, 14, 16, 17, 22$. We will not delve into the definition of the `intr` owner here but the interested reader can examine the definition of the `intr` step function provided in appendix B.

In order to facilitate the proof of (`cdeq+ <-> intr`), we decided to simplify the proof requirements into more direct steps. We first note that the theorems `labels-equal->>` and `well-founded->>` were easy to prove in each refinement proof we performed and so we will ignore them for the sake of

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*	

Figure 8: Comparing `cdeq+` thief with `intr` thief

presentation. The form of the theorem `rep-matches->>` is not conducive to ACL2 proof since the case analysis ACL2 performs is derived from whether or not `intr` can match the step of `cdeq+`. In order to better direct ACL2 to the desired result, we defined a predicate `commit` which takes a state and an input and returns `nil` if and only if `cdeq+` stutters. Using `commit` we can split the theorem `rep-matches->>` into three theorems which are easier to prove. We also defined a predicate `suff` which replaces `inv` in `rep-matches->>` and introduces the additional proof requirement that `inv` implies `suff`. The introduction of `commit` and `suff` results in splitting `rep-matches->>` into the following four theorems:

```
(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))
```

In most cases, the definition of `inv` is far more detailed and constraining (since it must persist) than needed in order to prove `rep-matches->>`. The predicate `suff` instead can be used to define the minimal assumptions required for proving `rep-matches->>` and further provides a starting point for the definition of `inv`. In fact, once the proper definitions for `rep`, `rank`, `commit`, `suff`, and `pick` were determined and some simple theorems about the variable translations were proven (e.g. `(equal (get-top (mend bot top mem)) (val (<- mem top)))`), the above theorems went through ACL2 with minimal assistance. In each of these theorems, ACL2 performed the necessary case-split (i.e. which process was selected, what is the current location, etc.) and simplified each case to `T`. Indeed, the difficulty of proving each refinement step correlated directly with the difficulty in coming up with the correct definitions for the witnessing functions and in proving `inv-persists->>`.

Subtle details would be exposed only during attempts at proving that the invariant persisted. For example, when the `cdeq+` thief is at *loc* 8, it is sufficient to assume:

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

when proving the theorems above. But in the invariant, we have to strengthen this to (where `age<<` is the lexicographic ordering of a pair of numbers):

```
(if (equal (age s) (old f))
    (= (xctr f) (xctr s))
  (and (age<< (old f) (age s))
       (< (xctr f) (xctr s))))
```

and in addition, this condition has to hold from thief *loc*s 2 through 8. While the definitions of `rank`, `rep`, `commit`, and `suff` for (`cdeq+ <-> intr`) were intuitive, the definition of `inv` was detailed and unintuitive. This makes the use of ACL2 that much more important since it lifts the analysis burden of a human peer from the understanding of the details of the definitions and proofs to the understanding of "what" was proven with an implicit trust that ACL2 is sound.

Where the nature of the refinement from `cdeq+` to `intr` was clear (i.e. hide "local" steps), the refinement from `intr+` to `spec` is a little more subtle. In particular, notice that in the `intr` thief function that it is possible for the thief to fail to pop the top of the DEQ (i.e. when $ctr \neq CTR$ at 2) even when the DEQ is non-empty. It was one of our goals in analyzing the concurrent deque to show that when the deque is non-empty, progress is made in popping items from the top of the deque. In order to ensure this, we defined `spec` such that thief steps cannot fail to pop the top of the deque. Thus, when a thief fails in `intr+`, `intr+` stutters and the `rank` for `intr+` states must decrease. Thus, there can only be a finite number of thief failures in-between two thief successes. The `rank` function for `intr+` is defined as follows:

```
(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)))))
```

When the deque is non-empty, the rank is a triple consisting of the owner's local rank, the expression `(miss-count (tvs st) (max-thf) (ctr (shr st)))` which counts up the number of thieves whose local *ctr* value differs from the global CTR value, and finally a summation of local ranks for each thief. Intuitively, any step the owner takes will either decrease its local rank or be matched by `spec`. When a thief takes a step, either the step is matched by `spec`, or the thief's local rank has decreased and its *ctr* hasn't changed, or the thief's local *ctr* has changed, but it is now equal to CTR. Thus, eventually every thief reaches the point where $ctr = CTR$ and this can only change if one of the processes succeeded in popping the top of the deque. Unfortunately, this rank does not decrease if the DEQ is empty (since CTR may not change) and this is why the `rank` function changes to a different measure when the deque is empty.

The remaining definitions and proofs of the requisite theorems for (`intr+ >> spec`) were easy to deduce from either intuition or failed ACL2 proofs. While some of the definitions were subtle, the relative simplicity of `intr+` in comparison to `cdeq+` significantly reduced the complexity of proving (`intr+ >> spec`) in comparison to (`cdeq+ <-> intr`). It is worth noting that `spec` uses an extra input field to determine if a thief can steal the last item from the owner. We added this extra input to `spec` in order to simplify the behavior of the thief steps, but this additional nondeterminism also meant that we could not prove (`intr+ <-> spec`). Since we use the `spec` function as the definition of "correct" behavior, we valued simplicity in the definition of `spec` more than the preservation of equivalence with `cdeq`.

5 Summary

In summary, we proved the following chain of refinements:

```
cdeq <-> cdeq+ <-> intr <-> intr+ >> spec
```

where `cdeq+` and `intr+` are simply `cdeq` and `intr`, respectively, with some additional auxiliary variables needed to complete the next step in the chain. The refinement steps (`cdeq+ <-> intr`) and (`intr+ >> spec`) allowed finite stuttering in `cdeq+` and `intr+` respectively which was important to enable systems at different levels of abstraction to be compared while ensuring progress. The most difficult step of these proofs was the definition of and persistence proof for the invariant for `cdeq+`. Although the invariant was detailed, once it was defined correctly ACL2 was able to verify it in very reasonable time. The longest proof time we encountered with ACL2 was for the theorem `thf-inv-onr-thm` which took ACL2 about two minutes to prove (without printing).

```
(defthm thf-inv-onr-thm       ;; in (cdeq+ <-> intr) refinement
  (implies (and (inv-shr s)
                (inv-onr o s)
                (assume-thf f s))  ;; implied by (inv-thf f s)
           (inv-onr o (c+-thf-s f s))))
```

This theorem states that the owner's invariant is preserved when a thief updates the shared variables. The main problem with interactively proving theorems of this nature is the turnaround time between submission and failure. This theorem, for instance, case splits into 1663 subgoals. On the one hand, it is desirable to inhibit the proof output of ACL2 in order to get a reasonable runtime on theorems with this many subgoals. On the other hand, when the theorem fails we need to examine the cases which failed and this, in turn, requires viewing the proof output. One solution to this quagmire is for the user to be extremely careful and diligent in the definition of the invariant and hope that the proof simply goes through. We do not recommend this approach because it is easy to add too many unnecesary details which complicate the user's definition task and ACL2's proof task. The approach we we took in cases such as this was to use the ACL2 proof checker. For intance, we used the following iterative ACL2 interaction cycle in order to find the proper definition of `inv-onr`:

```
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 failed 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 ...
```

It is also useful to define proof-checker macros which allow you to use a single command to perform a series of repetitive proof checker tasks. The interested reader should consult the ACL2 documentation of the proof checker and the special form `define-pc-macro`.

Overall, ACL2 was indeed a valuable tool in this proof effort. While it is possible to do proofs of correctness for concurrent programs by hand, the case analysis required may obfuscate human error or omission. Mechanical verification enforces a rigorous definition and proof which will likely cover cases that may otherwise be missed. In fact, an effective interaction with ACL2 was actually useful in iteratively refining the persistent invariant definitions. It seems unlikely that these invariants would have been as complete if it were not for ACL2's stubborness in accepting their correctness.

We conclude with a brief comparison between the proof of correctness presented in [3] and the proof presented here. The proof presented in [3] involves demonstrating that any interleaving behavior of the concurrent deque program can be transformed into a corresponding synchronous behavior (i.e. one where every process performs pushes and pops atomically). This transformation is carried out by permuting small sequences of program steps, termed *bursts*, of different processes until the behavior is synchronous. These permutations are presented through a series of sixteen congruences, beginning with the identity relation and concluding with a relation tieing every behavior to a synchronous behavior. For instance, assume we have two program bursts $B1; B2$ of processes $P1$ and $P2$, and each burst only modified local variables, then we could permute $B1$ and $B2$ without affecting the resulting behavior since we end up in the same resulting state.

The basic statement of correctness in [3] is elegant, but the analysis is tedious. An inherent benefit of mechanized proof, when it is practical, is that the burden of "belief" is lifted from the details of a proof to the understanding of the proof statement and a belief in the mechanized checker. Beyond this point, it is inherently more difficult to mechanically reason about behaviors or sequences of steps than it is to reason about single steps. All of the ACL2 theorems presented in this paper involve no more than a single step of at most two separate programs. In addition, the statement of stuttering refinement is clear and its application requires only an understanding of the `spec` program and the `label` function. This does necessitate the reader to mentally exercise the `spec` program to determine if it truly defines legal behaviors, and this, in turn, drives the need for the simplest `spec` possible. An additional benefit of the approach presented in this paper is the ability to demonstrate progress. The behavioral correspondence proved in [3] does not ensure progress to the next `pop-top` when the deque is non-empty. Indeed, stating this property would have further complicated the definition of a legal synchronous behavior. In our case, it is simply a result of proving a stuttering refinement and the observation that a thief must pop the deque when it is non-empty in the `spec` program.

5.1 Acknowledgements

We wish to thank Sandip Ray, Greg Plaxton, and Robert Blumhofe for posing the initial challenge of verifying their program and Sandip, in particular, for checking that the work presented here met their verification goals. We also want to acknowledge input received from Pete Manolios, J Moore, and Matt Kaufmann when an earlier version of this work was presented to them. We especially want to thank Pete for pushing the search for "the simplest specification program" and pointing out an error in an earlier version of the labeling function, and Matt Kaufmann who helped tremendously by answering questions about the proof checker and providing significant improvements to the records book which was used extensively in this work.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253-284, May 1991.

[2] N. Arora, R. Blumhofe, and C. Plaxton. Thread Scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1998.

[3] R. Blumofe, C. Plaxton, and S. Ray. Verification of a Concurrent Deque Implementation. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-99-11. June 1999.

[4] T. Henzinger, S. Qadeer, and S. Rajamani. Assume-guarantee refinement between different time scales. Proceedings of the 11th International Conference on Computer-aided Verification, Lecture Notes in Computer Science 1633, Springer-Verlag, 1999.

[5] P. Manolios. Correctness of Pipelined Machines, In W. Hunt, and S. Johnson, editors, *Formal Methods in Computer-Aided Design*, FMCAD 2000, LNCS. Springer-Verlag, 2000.

[6] P. Manolios, K. Namjoshi, R. Sumners. Linking theorem proving and model checking using well-founded bisimulation. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification*, volume 1633 of LNCS, Springer-Verlag, 1999.

[7] J. Moore. A Mechanically Checked Proof of a Multiprocessor Result via a Uniprocessor View. *Formal Methods in System Design*, 14(2), March, 1999, pp. 213-22

[8] K. Namjoshi. A Simple characterization of stuttering bisimulation. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of LNCS, 1997.

A Rewrite rules for record operations

```
;; (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-same-s
  (implies (force (fieldp a))
           (equal (g a (s a v r))
                  v)))

(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))))

(defthm s-same-g
  (implies (force (fieldp a))
           (equal (s a (g a r) r)
                  r)))

(defthm s-same-s
  (implies (force (fieldp a))
           (equal (s a y (s a x r))
                  (s a y r))))

(defthm s-diff-s
  (implies (and (force (fieldp a))
                (force (fieldp b))
                (not (equal a b)))
           (equal (s b y (s a x r))
                  (s a x (s b y r))))
  :rule-classes ((:rewrite :loop-stopper ((b a s)))))
```

B Definition of intermediate system

```
(defun i-thf-s (f s)
  (case (loc f)
        (2 (if (and (itm f)
                    (= (ctr f) (ctr s)))
               (>s :deq (drop-top (deq s))
                   :ctr (1+ (ctr s)))
             s))
        (3 (>s :ret (itm f) :clk (1+ (clk s))))
        (t s)))
```

```
(defun i-thf-f (f s)
  (case (loc f)
        ;; popTop
        (0 (>f :loc 1 :ctr (ctr s)))
        (1 (>f :loc 2 :itm (get-top (deq s))))
        (2 (>f :loc (if (and (itm f)
                             (= (ctr s) (ctr f)))
                        3 0)))
        (t (>f :loc 0))))

(defun i-onr-s (o s)
  (case (loc o)
        (1 (>s :deq (drop-bot (deq s))))
        (3 (if (or (= (ctr o) (ctr s))
                   (and (atom (deq s))
                        (implies (itm o) (ctr o))
                        (not (xzero o))))
               (>s :ctr (1+ (ctr s)))
             s))
        (4 (>s :ret (itm o) :clk (1+ (clk s))))
        (5 (>s :deq (push-bot (dtm o) (deq s))))
        (t s)))
                                                   ;;locations in cdeq owner
(defun i-onr-o (p d o s)                           ;;  bot = 0  |  bot > 0
  (case (loc o)                                    ;;  -------------------
        (0 (if p (>o :loc 5 :dtm d)                ;;     0     |    0
             (>o :loc 1)))                         ;;           |
        ;; popBottom
        (1 (>o :loc 2                              ;;     1     |    5
               :itm (get-bot (deq s))              ;;           |
               :ctr (and (one-eltp (deq s))        ;;           |
                         (ctr s))))                ;;           |
        (2 (let ((o (>o :loc 3)))                  ;;     2     |    7
             (cond ((or (not (itm o))              ;;           |
                        (consp (deq s))            ;;           |
                        (= (ctr o) (ctr s)))       ;;           |
                    o)                             ;;           |
                   ((not (ctr o)) (>o :ctr (ctr s))) ;;         |
                   (t (>o :itm nil)))))            ;;           |
        (3 (>o :loc (if (and (itm o)               ;;     3     |  8,14,17
                             (implies (ctr o)      ;;           |
                                      (= (ctr o) ;;             |
                                         (ctr s)))) ;;          |
                        4 0)                       ;;           |
               :xzero (implies (itm o) (ctr o))    ;;           |
               :ctr nil))                          ;;           |
    ;;  (4 (>o :loc 0))                            ;;           |   9,16
        ;; pushBottom
        (5 (>o :loc 0 :xzero nil))                 ;;    22     |    22
        (t (>o :loc 0))))
```

```
(DEFUN intr (st in)
  (let* ((ndx (thf (ndx in)))
         (dtm (dtm in))
         (psh (psh in))
         (tvs (tvs st))
         (shr (shr st))
         (onr (onr st))
         (thf (<- tvs ndx)))
    (if (ndx in)
        (>st :tvs (-> tvs ndx (i-thf-f thf shr))
             :shr (i-thf-s thf shr))
      (>st :onr (i-onr-o psh dtm onr shr)
           :shr (i-onr-s onr shr)))))
```