Fair Environment Assumptions in ACL2

ACL2 Workshop 2003

July 13, 2003

Rob Sumners

Advanced Micro Devices, Inc.

robert.sumners@amd.com

⌊ The Need for ***Fairness*** ⌋

- *reactive systems* are systems which maintain an ongoing interaction with an environment
  - Common examples: operating systems, concurrent algorithms, microprocessors, database transaction systems, etc.
- The specification of a reactive system will often include several *progress* properties
  - e.g. for a transaction system, every transaction eventually completes
- In order to prove progress for reactive systems, one often has to assume the environment makes "progress"
  - We term these progress assumptions *fair environment assumptions*

⌊ Simple Reactive System in ACL2 ⌋

- We assume a reactive system is defined in ACL2 using a binary `step` function and a constant `init` function
  - The `step` function takes the current state and an input from the environment and returns the next state
  - The `init` constant function returns the initial state of the system

- Consider the following simple reactive system:

```
(defun init () 0)

(defun step (s i)
  (let ((s (if (= s i) (1+ s) s)))
    (if (<= s (UB)) s 0)))
```

  - where `(UB)` is an arbitrary natural number Upper-Bound

⌊ Simple Progress Property in ACL2 ⌋

- Assume the following function:

```
(defun good (s) (= s (UB)))
```

- Consider the following *Progress* property:

  – At any time in any run of the system, `(good s)` will hold for some future state `s` in the run

- But, the system may get "stuck" if inputs are selected unfairly

  – Thus we need to assume fair selection of inputs in the statement of our property

⌊ Specifying Progress (and Fairness) ⌋

- In English: Assuming fair input selection, then at all times, eventually `(good s)`

- In (pseudo) LTL:

$$(\forall k \in \Phi : (GF(\texttt{i} = k))) \Rightarrow (GF(\texttt{good s}))$$

  - Φ is the *selection set* and in this example must include the natural numbers between 0 and `(UB)`

  - $GF \equiv$ *infinitely often*

- How do we specify this in ACL2?

  - The straightforward specification of progress (and fairness) involves statements about infinite sequences of states (and inputs)

  - But, in practice, we can reduce this to the definition and proofs of well-founded measures and invariants over single steps of the system

⌊ Specifying Progress in ACL2 ⌋

- In order to define progress, we need an infinite *run* of the system:

```
(encapsulate (((env *) => *)) ... )
;; arbitrary infinite input sequence

(defun run (n)
  (if (zp n) (init)
    (let ((n (1- n)))
      (step (run n) (env n)))))
```

- We define our progress property ($GF($`good s`$)$) using `defun-sk`:

```
(defun natp (x) (and (integerp x) (>= x 0)))

(defun time>= (y x)
  (and (natp y) (implies (natp x) (>= y x))))

(defun-sk eventually-good (x)
  (exists y (and (time>= y x) (good (run y)))))

(defthm progress (eventually-good n))
```

⌊ Specifying Fair Selection in ACL2 ⌋

- Approach #1: Define the notion of fair selection using **`defun-sk`** and add it as an hypothesis to the relevant theorems

```
(defun-sk exists-future (k x)
  (exists y (and (time>= y x)
                 (equal (env y) k))))

(defun-sk fair-selection ()
  (forall (k n) (exists-future k n)))
```

- Assuming **`(fair-selection)`**, we can now prove **`progress`**

```
(defthm progress
  (implies (fair-selection)
           (eventually-good n)))
```

  - In this case, Φ is the ACL2 universe

- But, how do we prove this?

⌊ Approach #1: Defining progress witness ⌋

- In order to prove `(eventually-good n)`, we define a witness function which returns the next time at which `good` will hold:

```
(defun good-time (n)
  (if (good (run n)) n (good-time (1+ n))))
```

- In order to admit `good-time`, we will need to define a measure
  - Assume `(fair-selection)` to define one component of the measure – `(env-measure k n)` – with the following property:

```
(defthm env-measure-property
  (and (natp (env-measure k n))
       (implies (and (fair-selection)
                     (natp n)
                     (not (equal (env n) k)))
                (< (env-measure k (1+ n))
                   (env-measure k n)))))
```

⌊ Approach #1: Admitting the witness ⌋

- We will need to modify the witness function:

```
(defun good-time (n)
  (declare (xargs :measure (good-measure n)))
  (cond ((not (fair-selection)) 0)
        ((not (natp n)) (good-time 0))
        ((good (run n)) n)
        (t (good-time (1+ n)))))
```

- Where the appropriate measure is defined by:

```
(defun good-measure (n)
  (lexprod
   (if (natp n) 1 2)
   (1+ (nfix (- (upper-bound) (run n))))
   (env-measure (run n) n)))
```

- A useful property of `good-time`:

```
(defthm good-of-good-time
  (implies (fair-selection)
           (good (run (good-time n)))))
```

⌊ Approach #1: Drawbacks ⌋

- The assumption of `(fair-selection)` implies the countability of the ACL2 universe
- Must include `(fair-selection)` as an hypothesis in several theorems
  - This inclusion follows a pattern and could be removed with a macro.
- Approach #2: Can we define an encapsulated fair environment on a subset Φ of the ACL2 universe?
  - Φ must be countable, but the larger Φ is, the better
- We factor this into two problems to solve:
  - Define a fair selector of the natural numbers
  - Define an invertible mapping from Φ into the naturals

⌊ Approach #2: Fair selection of naturals ⌋

- Problem: define `(env n)` and `(env-measure k n)` which satisfy:

```
(defthm env-measure-property
  (and (natp (env-measure k n))
       (implies (and (natp k)  ;; only change
                     (natp n)
                     (not (equal (env n) k)))
                (< (env-measure k (1+ n))
                   (env-measure k n)))))
```

- Solution: define a round-robin where the upper-bound on the cycle is always increasing

```
(defun fair-step (f)
  (let ((ctr (car f)) (top (cdr f)))
    (if (< ctr top)
        (cons (1+ ctr) top)
      (cons 0 (1+ top)))))

(defun fair-init () (cons 0 0))
```

⌊ Approach #2: Fair selection ... - 2 ⌋

- We can now define `env` and `env-measure` witness functions with the desired property:

```
(defun fair-run (n)
  (if (zp n) (fair-init)
    (fair-step (fair-run (1- n)))))

(defun env (n) (car (fair-run n)))

(defun fair-ctr (goal ctr top)
  (declare ...)
  (cond (... 0)
        ((equal ctr goal) 1)
        ((< ctr top)
         (1+ (fair-ctr goal (1+ ctr) top)))
        (t
         (1+ (fair-ctr goal 0 (1+ top))))))

(defun env-measure (k n)
  (fair-ctr k
            (car (fair-run n))
            (cdr (fair-run n))))
```

⌊ Approach #2: Transferring to Φ ⌋

- We define Φ to be the *nice* objects with the following recognizer:

```
(defun nicep (x)
  (or (stringp x)
      (characterp x)
      (acl2-numberp x)
      (symbolp x)
      (and (consp x)
           (nicep (car x))
           (nicep (cdr x)))))
```

- Define an invertible mapping to the natural numbers as the composition of:
  - An invertible mapping from *nice* objects into the *simple-trees*
  - An invertible mapping from the *simple-trees* into the naturals

- Transfer the fair selection of naturals to Φ using the mapping and its inverse appropriately

⌊ Approach #2: Application to Example ⌋

- Using the constrained fair selection of *nice* objects, we can now prove the theorems for our example without the `(fair-selection)` hypotheses:
  - For example, the following are now theorems:

```
(defthm good-of-good-time
  (good (run (good-time n)))))

(defthm progress (eventually-good n))
```

- If fair selection of the *nice* objects is sufficient (as in our example), then we recommend Approach #2
  - Otherwise, either use Approach #1 or use Approach #2 and maintain a redirection table in the system step function

⌊ Approach #2: More Complex Example ⌋

- A mutual exclusion protocol with the following `step` and `good` functions:

```
(defun step (s i)
  (if (prp i)
      (let* ((ndx (car s))
             (prs (cdr s))
             (p (getp i prs))
             (p+ (next-pc p))
             (p+ (if (and (in-crit p+)
                          (/= i ndx))
                     p
                   p+))
             (prs (setp i p+ prs))
             (n+ (next-pr ndx))
             (ndx (if (and (not (in-crit p+))
                           (= i ndx))
                        n+
                      ndx)))
        (cons ndx prs))
    s))

(defun good (s)
  (in-crit (getp (pick-pr) (cdr s))))
```

⌊ Approach #2: More Complex ... - 2 ⌋

- Good News: We only need to change the definition of `good-measure`
- Bad News:

```
(defun good-measure (n)
  (let* ((s (run n))
         (ndx (car s))
         (prs (cdr s))
         (nogo (not (equal ndx (pick-pr)))))
    (lexprod
     (if (natp n) 1 2)
     (nfix (- (crit-pc) (getp (pick-pr) prs)))
     (if nogo 2 1)
     (if nogo
         (if (> ndx (pick-pr))
             (+ (- (last-pr) ndx)
                (1+ (pick-pr)))
           (- (pick-pr) ndx))
       0)
     (if nogo
         (- (last-pc) (getp ndx prs))
       0)
     (env-measure ndx n))))
```

⌊ Further Extensions? ⌋

- Conditional Fairness:

  – We presented *unconditional* fairness, what about *conditional* fairness?

  – Imagine a predicate `(legal s i)` such that our `step` function was only defined for `legal` inputs at the current state

  – We would like to have a fair environment which ensured:

  $$\forall k \in \Phi : (GF(\texttt{legal s } k) \Rightarrow GF(\texttt{i} = k))$$

  – A *solution* to this problem is provided in the supporting materials, but its use is not recommended since it requires tighter composition between system and environment

- Real-time Constraints:

  – Some algorithms require bounds on the relative frequency of selections of different inputs in order to function

  – This is an area of future work

⌊ Summary and Conclusions ⌋

- We have presented two approaches to the use of fair environment assumptions in ACL2
  - One approach requires a `(fair-selection)` assumption, the other restricts the selection set to *nice* objects
- In practice, the example proofs of progress provide a template for proving progress for other systems
  - The definition of the function `good-measure` will be specific to a given system and will include the necessary calls of `env-measure`
- Related Work: Mechanization of UNITY in PC-NQTHM by D. Goldschlag
  - Work focuses more on the mechanization of UNITY proof rules (which rely on fairness) in PC-NQTHM rather than the definition of fair environments