Fair Environment Assumptions in ACL2

Rob Sumners

Advanced Micro Devices
robert.sumners@amd.com

Abstract

Liveness (or progress) proofs about reactive systems generally require assumptions ensuring the fair selection of particular components of the input to the reactive systems. We document our approach to defining these fair environments and their use in liveness proofs in the theorem prover ACL2.

1 Introduction

The term *reactive system* generally refers to any system (or design, program, etc.) which functions via an ongoing interaction with some environment. In ACL2, reactive systems are commonly defined by so-called *step* functions which are binary functions – `(step s i)` – that take the current state `s` of the *system* and an input `i` from the *environment* and return the next state of the system. A *behavior* or *run* of such a system is defined to be an infinite sequence of states (i.e. a function mapping naturals to states) as follows:

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

where `(init)` is some defined initial state of the system and `(env k)` is an arbitrary (e.g. defined via `encapsulate`) infinite sequence of inputs which provides the environment's input at each natural time `k`.[1]

The specifications of correct behavior for reactive systems generally consist of two types of properties, *safety* (i.e. nothing "bad" happens) and *liveness* (i.e. something "good" eventually happens)[1]. For example, if the system processed database transactions, then a safety property may state that "the results of each transaction is valid" where a liveness property may state that "every transaction eventually completes". In ACL2, proving safety properties generally involves proving an invariant of the reachable states which correlates to proving the theorem:

```
(defthm safety (not (bad (run k))))
```

where `bad` is a predicate defining illegal states (e.g. invalid transaction results). For liveness properties, the proof requirement is to show that eventually something "good" occurs, which in ACL2 is encoded by a `defun-sk`,

```
(defun-sk eventually-good (x) (exists (y) (and (time>= y x) (good (run y)))))
(defthm liveness (eventually-good k))
```

where `good` is a predicate defining a "good" event occurring at the state (e.g. the completion of a transaction). The theorem `liveness` states that at all times `k`, there exists some future (natural) time `k+` where `(good (run k+))`.

In the course of proving liveness properties, the need frequently arises to assume that the inputs to the system (or some component of the input to the system) is selected *fairly*. For example, in many cases, the system is an asynchronous composition of some number of component processes (i.e. the state of the system is a list of component process states). The step function for this type of system might be defined as follows:

[1] Because we need to state assumptions about the inputs selected by the environment, we need to model the environment via a function `env`. This necessity is unfortunate since it allows the possibility for a `step` function to "cheat" by calling `env` to peek at future inputs from the environment which is in violation of the modeling of nondeterministic choice by an environment. In practice, this can be avoided by requiring that the `step` function be defined before the `env` function – although we did not take this approach in the supporting materials for didactic reasons.

```
(defun select (i) (nfix (car i)))
(defun step (s i) (update-nth (select i) (step-process (nth (select i) s) s i) s))
```

where `step-process` is the step function for a single process state, and `select` is the component of the input which *selects* the process that will take the next step. A liveness property for this type of system may require that every process which starts some operation, eventually completes the operation which it began. One way in which this liveness property would be invalidated is if after a particular process started an operation, the process was never subsequently selected. Thus, a proper restatement of this property is "assuming fair selection of processes, then whenever a process starts an operation, the operation is eventually completed". Our goal is to provide effective and generally useful books for defining such fair environment assumptions. These books are included in the supporting materials of this paper and in the rest of this paper we will present an incrementally elaborate series of fair environments concluding with the instantiable fair environments included in the supporting materials.

2 Fair Environments

The first question we need to answer is what form of definition our fair environment assumptions should take. There are multiple facets to this question. First, we want to capture the intention of fair assumptions, but further, we want to assume nothing more than an environment which is fair. We will use ACL2's `encapsulate` feature to define functions with the minimal necessary constraints codified as theorems exported from the encapsulation.

Second, the theorems exported from the encapsulation need to be in a useful form. Several previous proof efforts with ACL2 have shown that an effective way to prove liveness in ACL2 is to define a measure function which maps states to ordinals and prove that either the predicate `good` is true or the measure decreases (very similar to the work on stuttering refinement proofs [4, 5, 3]). In this context of liveness proofs, the most useful format for a fairness assumption is to define a natural number measure for each possible input which decreases until that input is selected.

To these ends, we define a fair environment by two encapsulated functions `env` and `env-measure`. The function `env` is an infinite sequence of inputs (as before), and `env-measure` is a function which takes an input and a time and returns a measure which decreases strictly until the input is selected. In particular, we define a fair environment by the following encapsulation:

```
(defun natp (x) (and (integerp x) (>= x 0)))
(encapsulate (((env *) => *)
              ((env-measure * *) => *))

  (local (defun env (k) ...))
  (local (defun env-measure (i k) ...))

  (defthm environment-is-fair
    (and (natp (env-measure i k)))
         (implies (and (selectp i)
                       (natp k)
                       (not (equal i (env k))))
                  (< (env-measure i (1+ k))
                     (env-measure i k)))))
```

Where the unspecified function `selectp` is a recognizer for the set of inputs for which we wish to ensure fair selection. We provide a "proof" that the above definition of a fair environment is logically equivalent to a more direct declaration of a fair environment using `defun-sk` in the supporting materials.

In order to define the `local` witness functions for `env` and `env-measure` in the above encapsulation, we assume the definition of the following functions which define a *fair selector* state machine:

```
(fair-select f)
(fair-measure i f)
(fair-step f)
(fair-inv f)
(fair-init)
```

The function `fair-select` takes the current fair selector state `f` and returns the next input (i.e. the result of `env`). The function `fair-measure` takes an input `i` and the current fair selector state `f` and returns a natural number measure "bounding" the number of steps until the next selection of this input. The function `fair-step` takes the current fair selector state `f` and returns the next fair selector state. The predicate `fair-inv` is a good-state predicate on fair selector states which is guaranteed to be invariant over `fair-step` and true of the initial fair selector state `fair-init`. These functions are used to define the `env` and `env-measure` witness functions as follows:

```
(defun fair-run (k) (if (zp k) (fair-init) (fair-step (fair-run (1- k)))))
(local (defun env (k) (fair-select (fair-run k))))
(local (defun env-measure (i k) (fair-measure i (fair-run k))))
```

We demonstrate the idea of defining a fair environment by defining a fair selector which ensures fair selection of the natural numbers less than some upper bound. We will encapsulate this `upper-bound` as follows:

```
(encapsulate (((upper-bound) => *)) (local (defun upper-bound () 1))
  (defthm upper-bound-positive-natural
    (and (integerp (upper-bound)) (> (upper-bound) 0))))
```

The fair selector we chose to define for these bounded natural numbers is a simple "round-robin" scheduler which cycles the natural numbers less than `upper-bound`:

```
(defun selectp (i) (and (natp i) (< i (upper-bound))))
(defun fair-select (f) f)
(defun fair-measure (i f)
   (if (selectp i) (if (< i f) (+ i (- (upper-bound) f)) (- i f)) 0))
(defun fair-step (f) (let ((f (1+ f))) (if (< f (upper-bound)) f 0)))
(defun fair-inv (f) (selectp f))
(defun fair-init () 0)
```

The following pertinent properties about these fair-selector functions allow us to relieve the proof obligation of `environment-is-fair` for `env` and `env-measure`.

```
(defthm fair-measure-natural
  (implies (fair-inv f) (natp (fair-measure i f))))

(defthm fair-measure-decreases
  (implies (and (selectp i)
                (fair-inv f)
                (not (equal i (fair-select f))))
           (< (fair-measure i (fair-step f))
              (fair-measure i f))))

(defthm fair-inv-is-invariant
  (implies (fair-inv f) (fair-inv (fair-step f))))
```

This bounded fair selector may be of some use in its current form, but there are cases where we would like to have fair selection over an infinite set. For instance, this would allow fair environments for systems with an unbounded number of processes. Thus, we would like to have a fair selector over all of the natural numbers and not just those under an `upper-bound`. Fortunately, a fair selector for a set S is also a fair selector for any subset $X \subseteq S$ and thus, our goal should be to define a fair selector for as large a set S as possible.

We now consider the problem of defining a fair selector for the natural numbers. In the previous example, we used a round-robin approach to ensure fair selection. A similar approach will not work for all of the naturals since the naturals are not bounded. The solution in this case is to define a round-robin where the `upper-bound` is increased steadily such that all natural numbers are eventually less than the ever-increasing upper-bound. Further, all of the natural numbers which are less than the upper-bound will be selected again on the next cycle of the round-robin. We only include the `fair-select`, `fair-step`, and `selectp` functions for this extension below, but the complete definition and proofs are included in the supporting materials:

```
(defun selectp (i) (natp i))
(defun fair-select (f) (car f))
(defun fair-step (f)
  (let ((a (car f)) (d (cdr f)))
    (if (< a d) (cons (1+ a) d) (cons 0 (1+ d)))))
```

Given we have defined a fair selector over the natural numbers, the next logical question is "can we define a fair selection of the ACL2 universe?" (i.e. `(defun selectp (i) t)`). Fair selection of a set S ensures the existence of a mapping from the set S into the natural numbers, and thus ensures that the set S is countable. Since the ACL2 universe is open, there exist models which include uncountable sets (e.g. the real numbers). Thus, a fair selector over the ACL2 universe cannot be defined without restricting the ACL2 universe to be countable which would require axiomatic extension of the logic. It is possible to use `defun-sk` to define the property of fair selection of all objects and add this as an assumption to various theorems. This approach is provided in the supporting materials, but we do not recommend its use in general.

We can use the fair selector over the natural numbers to provide fair selection of any countable set if we are given an invertible mapping from this countable set into the natural numbers. For example, consider the following recognizer of the so-called "nice" objects:

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

In the supporting materials for this paper, we include the definition of functions `nice->nat` and an inverse `nat->nice` which satisfy the following theorem:[2]

```
(defthm nice->nat->nice-properties
  (and (natp (nice->nat x))
       (nicep (nat->nice x))
       (implies (nicep x) (equal (nat->nice (nice->nat x)) x))))
```

We use these functions to "transfer" the fair selection of the natural numbers to the fair selection of "nice" objects (i.e. `(defun selectp (i) (nicep i))`) which include the naturals as a subset and as such we can simply use the fair selection of "nice" objects subsequently.

In order to demonstrate the application of these fair environment assumptions, we consider a simple counting system and prove a liveness property. The simple system and liveness property are defined by the following `step`, `init`, and `good` functions:

[2]Due to a technical limitation in ACL2 v2-7 and previous versions, it is impossible to define an inverse `nat->nice` which can construct any symbol – the package must be defined first with a `defpkg` event. This issue will be remedied in ACL2 v2-8, and thus, the definition of `nicep` in the supporting materials will be different in pre-v2-8 ACL2's with booleans and keywords as the only "nice" symbols.

```
(defun init () 0)
(defun step (s i)
  (let ((s (if (= s i) (1+ s) s)))
    (if (<= s (upper-bound)) s 0)))
(defun good (s) (= s (upper-bound)))
```

The key requirement in proving `liveness` is to define a function which provides a witness for the existential quantifier in the function `eventually-good`. The function in our example is the function `good-time`:

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

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

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

Intuitively, the function `good-time` returns the next time at which `good` will be true. This is accomplished by incrementing a time parameter until `good` is true. Thus, the question of `liveness` has been reduced to providing a measure for the termination of `good-time`. This measure will be composed of measures from the system being verified and various calls of the `env-measure` function to allow "stuttering" until the next key input is selected by the environment. In our example, the simple system consists of a counter which is cyclically incremented up-to `(upper-bound)` but it only increments if the external input equals the current counter. The measure for `good-time` is the lexicographic product of three terms. The term `(if (natp n) 1 2)` is included simply to pass over the case where an invalid time parameter is provided. The term `(1+ (nfix (- (upper-bound) (run n))))` is the number of counter increments before `good` is true. Finally, the term `(env-measure (run n) n)` bounds the number of inputs selected before a counter increment will occur.

Thus, the `environment-is-fair` property is utilized in the termination proof for the function `good-time`. The properties of the function `good-time` then allow us to prove `liveness`. While we do not claim that our reduction from proving `liveness` to admitting `good-time` is a general solution in its current form, it is likely the case that any proof of `liveness` would take a similar form. Further, it is our experience that determining the proper measure function in `liveness` properties for real systems is fairly straightforward and the majority of the proof effort is in defining and proving an inductive invariant which ensures the system component of the termination measure decreases with each step.

3 Conditional Fairness

The fair selectors we defined in the previous section afforded *unconditional* fairness whereby every element from the selection set was ensured to be selected at some point in the future irregardless of the system state. In some cases, it may be useful to restrict the set of legal inputs for the system in order to simplify the definition of the system. A naive approach to handle this case would be to take the input selected from a unconditional fair selector and "coerce" the input to be a legal input. The problem with this naive approach is that a particular input may only be selected when it is illegal and this would not be "fair" if the input were legal infinitely often. In order to support this case, we need to define a fair environment which provides a form of *conditional fairness* in which the environment assures that if an input is infinitely often *legal*, then it is infinitely often *selected*.

In this case, we need to define the fair environment in terms of a `legal-input` predicate which takes a system state and input and returns `T` if the input is legal to apply at the state. Further, we need to add a system state component to the fair selector functions in order to allow the fair selector to adjust its state according to which inputs are legal at a given system state. This complication breaks the nice functional composition of `fair-run`, `env`, and `run` functions from the previous section and which in turn, requires us to "move" the encapsulation to the fair selector. Due to these complications, we do not suggest the use of this conditional fair environment in general. In some cases though, conditional fairness may be required and thus we provide a "solution".

The properties we proved for the previous unconditional fair selectors are inappropriate in the case of conditional fairness. We need to adjust the decrements of the `fair-measure` such that (a) when an input is legal and not selected, the measure decreases, and (b) when an input is not selected, the measure does not increase. This is codified by the following theorem:

```
(defthm fair-measure-decreases
  (let ((m (fair-measure i f))
        (m+ (fair-measure i (fair-step f s x))))
    (implies (and (fair-inv f)
                  (nicep i)
                  (not (equal (fair-select f s) i)))
             (and (<= m+ m)
                  (implies (legal-input s i) (< m+ m))))))
```

In addition to the pair of natural numbers used in the previous unconditional fair selector, the conditional fair selector must maintain a list of inputs which were selected but were not legal when they were selected. In each `fair-step`, this list of bypassed inputs is first consulted to determine if an input which was illegal when selected, is now legal and if so, it is selected and removed from the list of bypassed inputs. If no bypassed input is now legal, then we continue the increasing round-robin. Another complication arises due to the need to satisfy the following additional theorem in the conditional fair selector encapsulation:

```
(defthm fair-select-must-be-legal
  (implies (fair-inv f) (legal-input s (fair-select f s))))
```

In order to satisfy this theorem, we need to continue "running" the increasing round-robin until we find an input which is legal for the current state (while collecting any inputs which are not legal in the bypassed input list). In order to satisfy these constraints, we assume a function `legal-witness` ensuring the existence of some legal input for every state:

```
(defthm legal-witness-is-legal-input
  (legal-input s (legal-witness s)))
```

Given these parameters and referring to the intuition before, we now define our `fair-step` function (the other functions for this conditional fair selector are included in the supporting materials):

```
(defun step-env (s bypass top ctr)
  (cond ((not (and (natp ctr) (natp top) (<= ctr top)))
         (list bypass 1 0))
        ((legal-input s (nat->nice ctr))
         (cons bypass
               (if (= top ctr) (list (1+ top) 0) (list top (1+ ctr)))))
        ((< ctr top)
         (step-env s (append bypass (list ctr)) top (1+ ctr)))
        (t
         (step-env s (append bypass (list top)) (1+ top) 0))))
```

```
(defun fair-step (f s x) (declare (ignore x))
  (let ((bypass (first f))
        (top    (second f))
        (ctr    (third f)))
    (let ((ndx (legal-in-list s bypass)))
      (if ndx
          (list (drop-from-list bypass ndx) top ctr)
        (step-env s bypass top ctr)))))
```

4 Conclusion

We presented a method (with definition and theorem support) for proving liveness properties in ACL2 requiring fair environment assumptions. We defined both unconditional and conditional fair selectors for the "nice" ACL2 objects (the countable set of constructible objects from the ACL2 universe). We also demonstrated the use of the unconditional fair selector in a simple but illustrative example.

This work provides no support for so-called "real-time" constraints whereby certain relative bounds for the rate of selection for different inputs can be assumed and used in the definition of a system. Our primary concern has been with capturing assumptions consistent with asynchronous compositions of processes and basic usage models. The consideration of real-time constraints is an area of future work.

References

[1] B. Alpern, F. Schneider. Recognizing safety and liveness. In *Distributed Computing*, 2:117–126, 1987.

[2] D. Goldschlag. A Mechanization of Unity in PC-NQTHM-92. In *Journal of Automated Reasoning*, 23(3), pp. 445-498, 1999.

[3] 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.

[4] 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.

[5] R. Sumners. An Incremental Stuttering Refinement Proof of a Concurrent Program in ACL2, in *Second ACL2 Workshop*, October 2000.