Fourth Halmstad Summer School on Testing
June 9-12, 2014

Testing and Verification in ACL2

Rex Page, University of Oklahoma

June 12, 9:30 – 10:30

What is ACL2 ?

ACL2 — A Computational Logic for Applicative Common Lisp
- ✓ programming language and mechanized logic
  - dialect of Common Lisp
  - conservative logic for effective mechanization/automation
  - 1st-order logic, terminating functions, no mutable variables
- ✓ Boyer/Moore theorem-prover, latest edition
  - 40-year history, proof checking and proof generation
  - maintenance and enhancement ongoing

Proof Pad — IDE for ACL2 (Eggensperger)
- ✓ programming + theorem proving = full ACL2
- ✓ + property-based testing
  - … but no narrowing, no user-defined data generators
- ✓ three other ACL2 IDEs
  - DrACuLA (DrRacket plug-in: PLT, Felleisen, Eastlund)
  - Emacs (comes with ACL2 installation) – ACL2 pros use this
  - ACL2 Sedan (Eclipse plug-in)

no property testing

Why ACL2 ?

instead of Isabelle or Coq or …

Learning curve
- ✓ students can succeed early with ACL2

Focus on software from the outset
- ✓ programming language a central element (Lisp)
- ✓ embedded logic for specifying software properties

Automated reasoning
- ✓ effective automation of property verification
  (full automation impossible of course … Church/Turing)
- ✓ makes ACL2 attractive for industrial apps
  AMD, Centaur/VIA Tech, Rockwell Collins, NSA, …

Goal

software that meets expectations

Specifying expectations
- ✓ suite of test cases
  input, expected output (operation x^2): $2^2 = 4$, $3^2 = 9$ (input, output)
- ✓ corner cases
  carefully chosen input, expected output: $(-1)^2 = 1$
- ✓ operator relationships
  Boolean formulas: $x^2 \geq 0$, $(x + y)^2 = x^2 + 2xy + y^2$, …

We expect tests to be theorems
- ✓ test cases: theorems with one-element domains
- ✓ operator relationships: more general theorems
- ✓ theorems: a familiar notion in software
  (eg, type checking proves theorems about types)

Goal for today

basic overview of ACL2 capability

Brief, introductory experience with ACL2
- ✓ specify and run tests of properties of operations
- ✓ verify some of those properties with proofs

ACL2's potential for software verification
- ✓ kinds of things that have been done with ACL2
- ✓ investment required for property verification

Focus on properties with large domains

Boolean formulas specify expectations

Example: sum and reverse operators
- ✓ $sum[x_1\ x_2\ \ldots\ x_n] = x_1 + x_2 + \ldots + x_n$
- ✓ $reverse[x_1\ x_2\ \ldots\ x_n] = [x_n\ x_{n-1}\ \ldots\ x_1]$

Expectation
- ✓ $sum[x_1\ x_2\ \ldots\ x_n] = sum(reverse[x_1\ x_2\ \ldots\ x_n])$

ACL2 spec for this expectation

```
(defthm sum=sum-reverse
 (= (sum xs)
    (sum(reverse xs))))
```

- prefix notation
- fully parenthesized

Proof Pad spec for testing the expectation

```
(defproperty test:sum=sum-reverse
 (xs :value (random-list-of (random-integer)))
 (= (sum xs)
    (sum(reverse xs))))
```

demo sum

Theorems are derived from axioms

axioms

x + 0 = x	{+ identity}
(-x) + x = 0	{+ complement}
x × 1 = x	{× identity}
x × 0 = 0	{× null}
x + y = y + x	{+ commutative}
x + (y + z) = (x + y) + z	{+ associative}
x × (y + z) = (x × y) + (x × z)	{distributive law}

x, y, z stand for any formula

ACL2 software
- ✓ program = system of equations
- ✓ constrains programs, but not computations (Turing complete)

Software Verification
- ✓ axioms = programs
- ✓ theorems = properties of programs

theorem: (-1)×(-1) = 1

derivation

(-1) × (-1)	
= ((-1) × (-1)) + 0	{+ id}
= ((-1) × (-1)) + ((-1) + 1)	{+ comp}
= (((-1)×(-1)) + (-1)) + 1	{+ assoc}
= (((-1)×(-1)) + (-1)×1) + 1	{× id}
= ((-1)×((-1) + 1)) + 1	{dist law}
= ((-1)×0) + 1	{+ comp}
= 0 + 1	{× null}
= 1 + 0	{+ comm}
= 1	{+ id}

Some ACL2 operations (informal specs)

algebra of software

(cons x [x_1 x_2 … x_n]) = [x x_1 x_2 … x_n]
(first [x_1 x_2 … x_{n+1}]) = x_1
(rest [x_1 x_2 … x_{n+1}]) = [x_2 … x_{n+1}]
(append [x_1 x_2 … x_n] [y_1 y_2 … y_m]) = [x_1 x_2 … x_n y_1 y_2 … y_m]
(len [x_1 x_2 … x_n]) = n

Axioms (equations of computation … programs)

(first (cons x xs)) = x	{first}
(rest (cons x xs)) = xs	{rest}
(append nil ys) = ys	{app0}
(append (cons x xs) ys) = (cons x (append xs ys))	{app1}
(len nil) = 0	{len0}
(len (cons x xs)) = 1 + (len xs)	{len1}

Theorems (properties derivable from axioms)

(append xs (append ys zs)) = (append (append xs ys) zs)	{app-assoc}
(len (append xs ys)) = (len xs) + (len ys)	{app-len}

Equations provide framework

axioms, theorems, programs, properties … formulated as equations

Properties (tests) that act as definitions

Two properties of concatenation

(append nil ys) = ys	{app0}
(append (cons x xs) ys) = (cons x (append xs ys))	{app1}

These equations have the following attributes
- ✓ comprehensive — all cases covered by left-hand-side formulas
- ✓ consistent — no two equations specify conflicting results
- ✓ computational

Properties (tests) that act as definitions

Two properties of concatenation

(append nil ys) = ys	{app0}
(append (cons x xs) ys) = (cons x (append xs ys))	{app1}

These equations have the following attributes
- ✓ comprehensive — all cases covered
- ✓ consistent — no two equations specify conflicting results
- ✓ computational
  circular reference on right-hand-side of equation closer to non-circular case than reference on left-hand-side

Properties (tests) that act as definitions

Two properties of concatenation

(append nil ys) = ys	{app0}
(append (cons x xs) ys) = (cons x (append xs ys))	{app1}

These equations have the following attributes
- ✓ comprehensive — all cases covered
- ✓ consistent — no two equations specify conflicting results
- ✓ computational
  circular reference on right-hand-side of equation closer to non-circular case than reference on left-hand-side

the 3 c's

Equations satisfying "the 3 c's" define an operator
- ✓ all properties of the operator derive from the equations
- ✓ including computational properties

Formal (ACL2) definition of concatenation

```
(defun append (xs ys)
  (if (consp xs) ; cons predicate: is xs non-empty?
      (cons (first xs) (append (rest xs) ys))  ; {app1}
      ys                                       ; {app0}
```


Formal specifications of properties (Proof Pad)

Some expected properties (informal specs)

(append xs (append ys zs)) = (append (append xs ys) zs)	{app-assoc}
(len (append xs ys)) = (len xs) + (len ys)	{app-len}

Formal specifications of properties (Proof Pad)

(append xs (append ys zs)) = (append (append xs ys) zs) {app-assoc}
(len (append xs ys)) = (len xs) + (len ys) {app-len}

informal

formal property

```
(defproperty test-app-assoc ; {app-assoc}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer))
   zs :value (random-list-of (random-integer)))
  (equal (append xs (append ys zs))
         (append (append xs ys) zs)))
```


Formal specifications of properties (Proof Pad)

(append xs (append ys zs)) = (append (append xs ys) zs) {app-assoc}
(len (append xs ys)) = (len xs) + (len ys) {app-len}

informal

formal properties (Proof Pad)

```
(defproperty test-app-assoc ; {app-assoc}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer))
   zs :value (random-list-of (random-integer)))
  (equal (append xs (append ys zs))
         (append (append xs ys) zs)))

(defproperty test-append-preserves-len ; {app-len}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (= (len (append xs ys)) (+ (len xs) (len ys))))
```


Formal specifications of theorems (ACL2)

(append xs (append ys zs)) = (append (append xs ys) zs) {app-assoc}
(len (append xs ys)) = (len xs) + (len ys) {app-len}

informal

properties (Proof Pad)

```
(defproperty test-app-assoc ; {app-assoc}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer))
   zs :value (random-list-of (random-integer)))
  (equal (append xs (append ys zs))
         (append (append xs ys) zs)))
(defthm app-assoc ; {app-assoc}
    (equal (append xs (append ys zs))
           (append (append xs ys) zs)))
(defproperty test-append-preserves-len ; {app-len}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (= (len (append xs ys)) (+ (len xs) (len ys))))
(defthm append-preserves-len ; {app-len}
  (= (len (append xs ys)) (+ (len xs) (len ys))))
```

demo app

theorems (ACL2)

Additional properties of concatenation

```
(defproperty test-app-suffix ; {app-sfx}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (equal (nthcdr (len xs) (append xs ys))
         ys))
(defproperty test-app-prefix ; {app-pfx}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (equal (prefix (len xs) (append xs ys))
         xs))
```

demo app-sfxpfx

Axioms defining prefix operator

(prefix 0 xs) = nil {pfx0 a}
(prefix n nil) = nil {pfx0 b}
(prefix (+ n 1) (cons x xs)) = (cons x (prefix n xs)) {pfx1}

Additional properties of concatenation

```
(defproperty test-app-suffix ; {app-sfx}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (equal (nthcdr (len xs) (append xs ys))
         ys))
(defproperty test-app-prefix ; {app-pfx}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (equal (prefix (len xs) (append xs ys))
         xs))
```

demo app-sfxpfx Whoops!

Axioms defining prefix operator

(prefix 0 xs) = nil {pfx0 a}
(prefix n nil) = nil {pfx0 b}
(prefix (+ n 1) (cons x xs)) = (cons x (prefix n xs)) {pfx1}

Additional properties of concatenation

```
; import some theorems of algebra
(include-book "arithmetic-3/top" :dir :system)
(defproperty test-app-suffix ; {app-sfx}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (equal (nthcdr (len xs) (append xs ys))
         ys))
(defproperty test-app-prefix ; {app-pfx}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (equal (prefix (len xs) (append xs ys))
         xs))
```

demoB app-sfxpfx

theorems of algebra needed for these proofs

Additional properties of concatenation

```
; import some theorems of algebra
(include-book "arithmetic-3/top" :dir :system)
(defproperty test-app-suffix ; {app-sfx}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (equal (nthcdr (len xs) (append xs ys))
         ys))
(defproperty test-app-prefix ; {app-pfx}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (implies (true-listp xs) ; xs must be a list
           (equal (prefix (len xs) (append xs ys))
                  xs)))
```

demoC app-sfxpfx

Practice exercises

Ex 1: Get ACL2 to prove the following property

```
(defproperty test-sum=sum-reverse
  (xs :value (random-list-of (random-integer)))
  (= (sum xs)
     (sum(reverse xs))))
```

Hints

1. You will need to define "sum"
2. The intrinsic "reverse" is tail-recursive, with accumulator, which complicates reasoning. Define "rev" as an append of the first element to the rev of the rest of the list.
3. Restate the property with "rev" and get ACL2 to prove it

Ex 2: Fast reverse: Use tail recursion to define "rev-app" so that is has the following {rev-app} property

(rev-app *xs* ys) = (append (rev xs) ys) {rev-app}

Ex 3: Get ACL2 to prove the rev-app equation

Notes: http://ceres.hh.se/mediawiki/index.php/HSST_2014

Install Proof Pad: http://proofpad.org

Ordered merge

(mrg [x_1 x_2… x_n] [y_1 y_2 … y_m]) = [z_1 z_2 … z_{n+m}]
where $z_1 \le z_2 \le \ldots \le z_{n+m}$ if
if $x_1 \le x_2 \le \ldots \le x_n$ and $y_1 \le y_2 \le \ldots \le y_m$

definitional properties of mrg

(mrg (cons x xs) (cons y ys)) = (cons x (mrg xs (cons y ys)))	{x≤y}
(mrg (cons x xs) (cons y ys)) = (cons y (mrg (cons x xs) ys))	{y<x}
(mrg xs nil) = xs	{mg0}
(mrg nil ys) = ys	{mg1}

Ordered merge

(mrg [x_1 x_2… x_n] [y_1 y_2 … y_m]) = [z_1 z_2 … z_{n+m}]
where $z_1 \le z_2 \le \ldots \le z_{n+m}$ if
if $x_1 \le x_2 \le \ldots \le x_n$ and $y_1 \le y_2 \le \ldots \le y_m$

definitional properties of mrg

(mrg (cons x xs) (cons y ys)) = (cons x (mrg xs (cons y ys)))	{x≤y}
(mrg (cons x xs) (cons y ys)) = (cons y (mrg (cons x xs) ys))	{y<x}
(mrg xs nil) = xs	{mg0}
(mrg nil ys) = ys	{mg1}

formal definition

```
(defun mrg (xs ys)
  (if (and (consp xs) (consp ys))
      (let* ((x (first xs)) (y (first ys)))
        (if (<= x y)
            (cons x (mrg (rest xs) ys))   ; {mgx}
            (cons y (mrg xs (rest ys))))) ; {mgy}
      (if (not (consp ys))
          xs          ; ys is empty       ; {mg0}
          ys)))       ; xs is empty       ; {mg1}
```


Ordered merge

(mrg [x_1 x_2… x_n] [y_1 y_2 … y_m]) = [z_1 z_2 … z_{n+m}]
where $z_1 \le z_2 \le \ldots \le z_{n+m}$ if
if $x_1 \le x_2 \le \ldots \le x_n$ and $y_1 \le y_2 \le \ldots \le y_m$

definitional properties of mrg

(mrg (cons x xs) (cons y ys)) = (cons x (mrg xs (cons y ys)))	{x≤y}
(mrg (cons x xs) (cons y ys)) = (cons y (mrg (cons x xs) ys))	{y<x}
(mrg xs nil) = xs	{mg0}
(mrg nil ys) = ys	{mg1}

One of our expectations is

- ✓ (up(mrg xs ys)) if (up xs) and (up ys) where
  (up[x_1 x_2… x_n]) = $x_1 \le x_2 \le \ldots \le x_n$

demo 4

Ordered merge

(mrg [x_1 x_2… x_n] [y_1 y_2 … y_m]) = [z_1 z_2 … z_{n+m}]
where $z_1 \le z_2 \le \ldots \le z_{n+m}$ if
if $x_1 \le x_2 \le \ldots \le x_n$ and $y_1 \le y_2 \le \ldots \le y_m$

definitional properties of mrg

(mrg (cons x xs) (cons y ys)) = (cons x (mrg xs (cons y ys)))	{x≤y}
(mrg (cons x xs) (cons y ys)) = (cons y (mrg (cons x xs) ys))	{y<x}
(mrg xs nil) = xs	{mg0}
(mrg nil ys) = ys	{mg1}

formal definition

```
(defun mrg (xs ys)
  (declare (xargs :measure (+ (len xs) (len ys))))
  (if (and (consp xs) (consp ys))
      (let* ((x (first xs)) (y (first ys)))
        (if (<= x y)
            (cons x (mrg (rest xs) ys))   ; {mgx}
            (cons y (mrg xs (rest ys))))) ; {mgy}
      (if (not (consp ys))
          xs          ; ys is empty       ; {mg0}
          ys)))       ; xs is empty       ; {mg1}
```

suggested induction scheme

Merge-sort

(msort $[x_1\ x_2 \ldots\ x_n]$ = $[z_1\ z_2\ \ldots\ z_n]$
where $z_1 \le z_2 \le \ldots \le z_n$
and $[z_1\ z_2\ \ldots\ z_n]$ is a permutation of = $[z_1\ z_2\ \ldots\ z_n]$

definitional properties of merge-sort

(msort [ ]) = [ ] {ms0}
(msort [x]) = [x] {ms1}
(msort $[x_1\ x_2 \ldots\ x_n]$) = (mrg (msort $[x_1\ x_2 \ldots\ x_{\lceil n/2 \rceil}]$ {ms2}
(msort $[x_{\lceil n/2 \rceil+1} \ldots\ x_n]$))

One of our expectations is
✓ (up(msort xs))

demo 5

any split putting half of the elements in one list and half in the other is okay

Merge-sort

(msort $[x_1\ x_2 \ldots\ x_n]$ = $[z_1\ z_2\ \ldots\ z_n]$
where $z_1 \le z_2 \le \ldots \le z_n$
and $[z_1\ z_2\ \ldots\ z_n]$ is a permutation of = $[z_1\ z_2\ \ldots\ z_n]$

definitional properties of merge-sort

(msort [ ]) = [ ] {ms0}
(msort [x]) = [x] {ms1}
(msort $[x_1\ x_2 \ldots\ x_n]$) = (mrg (msort $[x_1\ x_2 \ldots\ x_{\lceil n/2 \rceil}]$ {ms2}
(msort $[x_{\lceil n/2 \rceil+1} \ldots\ x_n]$))

One of our expectations is
✓ (up(msort xs))

maybe ACL2 doesn't know dmx reduces the list length

```
(defthm dmx-reduces-len-thm ; lemma, msort termination
   (implies (consp (rest xs))
            (and (< (len(first(dmx xs))) (len xs))
                 (< (len(second(dmx xs))) (len xs)))))
```


Merge-sort, with termination lemma

```
(defthm dmx-reduces-len ; lemma, msort termination
   (implies (consp (rest xs))
            (and (< (len(first(dmx xs))) (len xs))
                 (< (len(second(dmx xs))) (len xs)))))

(defun msort (xs)
  (declare  ; suggest using lemma to prove termination
   (xargs :measure (len xs)
          :hints (("Goal"
            :use ((:instance dmx-reduces-len))))))
  (if (consp(rest xs)) ; (len xs) > 1?
      (let* ((odds-evens (dmx xs)) ; xs = [x1 x2 ...]
             (odds       (first odds-evens))
             (evns       (second odds-evens)))
        (mrg (msort odds) (msort evns))) ;    {ms2}
      xs))                ; xs = [x1] or empty {ms1}
```


The End

June 12, 2014
9:30-10:30 session