Functional Programming and Theorem Proving for Undergraduates
A Progress Report

Carl Eastlund and Matthias Felleisen
Northeastern University

Rex Page
University of Oklahoma
History

- **Before 2003**
  - Traditional SE at OU (2-course sequence, 4th yr)
  - Process  Design  Testing/Validation
  - 60%  20%  20%

- **2003-2005**
  - SE course using ACL2 (FDPE 2005 report)
  - Process  Design  Testing/Validation
  - 30%  35%  35%
  - Successful despite crude programming env

- **2006 - present**
  - SE course with Dracula/ACL2 environment
  - 1st year course at NU using Dracula/ACL2
Mantra

- **Before 2003**
  - Traditional SE at OU (2-course sequence, 4th yr)
  - Process Design Testing/Validation
  - 60% 20% 20%
  - Process using ACL2 (FDPE 2005 report)
  - Design Testing/Validation
  - 30% 35% 35%
  - Successful despite crude programming env

- **2006 - present**
  - SE course with Dracula/ACL2 environment
  - 1st year course at NU using Dracula/ACL2

*Engineering is the application of principles of science and mathematics to the design of useful things.*
ACL2

;;; sqr : Int -> Int
(defun sqr (x)
  (* x x))

;;; All squares are nonnegative.
(defthm sqr>=0
  (implies (integerp x)
    (>= (sqr x) 0)))
ACL2

ACL2 Version 3.3. Level 1. Cbd "/Users/cce/Desktop/"

Distributed books directory "/Users/cce/Local/ACL2/3.3
/openmcl32/books/".
Type :help for help.
Type (good-bye) to quit completely out of ACL2.

ACL2 !>

*shell* Bot (25,7) (Shell:run)----6:25AM--
ACL2

;;; sqr : Int -> Int
(defun sqr (x)
  (* x x))

;;; All squares are nonnegative.
(defun sqr>=0
  (implies (integerp x)
    (>= (sqr x) 0)))
ACL2
Dracula

---

```
;; sqr : Int -> Int
(defun sqr (x)
  (* x x))

;; All squares are nonnegative.
(defthm sqr>=0
  (implies (integerp x)
           (>= (sqr x) 0)))
```

Welcome to DrScheme, version 4.1 [3m].
Language: Dracula v4.2.
> (sqr 2)
4
>
Dracula

;; short-worm : Integer Integer Direction -> Worm
;; Produces a worm without a tail.
(defun short-worm (x y d)
  (make-worm d (make-point x y) nil))

;; worm-turn : Worm Direction -> Worm
;; Changes the direction a worm faces.
(defun worm-turn (w d)
  (make-worm d (worm-head w) (worm-tail w)))

;; worm-move : Worm -> Worm
;; Moves the worm in the direction it faces.
(defun worm-move (w)
  (make-worm
   (worm-dir w)
   (point+ (worm-head w)
      (dir-delta
        (worm-dir w)))
   (drop-last-point
    (cons (worm-head w)
       (worm-tail w)))))
Dracula

;;; short-worm : Integer Integer Direction -> Worm
;;; Produces a worm without a tail.
(defun short-worm (x y d)
  (make-worm d (make-point x y) nil))

;;; worm-turn : Worm Direction -> Worm
;;; Changes the direction a worm faces.
(defun worm-turn (w d)
  (make-worm d (worm-head w) (worm-tail w)))

;;; worm-move : Worm -> Worm
;;; Moves the worm in the direction it faces.
(defun worm-move (w)
  (make-worm
    (worm-dir w)
    (point+ (worm-head w)
              (dir-delta
               (worm-dir w)))
    (drop-last-point
     (cons (worm-head w)
           (worm-tail w))))

Functional Programming and Theorem Proving for Undergraduates
FDPE 2008 - Rex Page / Carl Eastlund / Matthias Felleisen 10
Dracula

;;; short-worm : Integer Integer Direction -> Worm
;;; Produces a worm without a tail.
(defun short-worm (x y d)
  (make-worm d (make-point x y) nil))

;;; worm-turn : Worm Direction -> Worm
;;; Changes the direction a worm faces.
(defun worm-turn (w d)
  (make-worm d (worm-head w) (worm-tail w)))

;;; worm-move : Worm -> Worm
;;; Moves the worm in the direction it faces.
(defun worm-move (w)
  (make-worm
    (worm-dir w)
    (point+ (worm-head w)
      (dir-delta (worm-dir w)))
    (drop-last-point
      (cor ...))
    nil))
Dracula

```
;; short-worm : Integer Integer Direction -> Worm
;; Produces a worm without a tail.
(defun short-worm (x y d)
  (make-worm d (make-point x y) nil))

;; worm-turn : Worm Direction -> Worm
;; Changes the direction.
(defun worm-turn (worm dir)
  (make-worm dir worm
  (drop-last-pair (cons (worm dir)
  (point+ (worm dir)
  (dir
  (worm dir)
  (drop-last-pair (cons (worm dir)
  (point+ (worm dir)
  (dir
  (worm dir)

2.4.8 Lists

List functions can also be found in the sections on Association Lists, Sets, and Sequences.

---

Examples:
1. `(cons 1 nil)`
2. `(cons 2 (cons 3 nil))`
3. `(cons 4 5)`
4. `4 . 5)`
5. `(cons x)`
```
Dracula

;; short-worm : (Worm) -> Worm
;; Produces a worm without a tail.
(defun short-worm (x y d)
  (make-worm d (make-point x y) nil))

;; worm-turn : Worm Direction -> Worm
;; Changes the direction a worm faces.
(defun worm-turn (w d)
  (make-worm d (worm-head w) (worm-tail w)))

;; worm-move : Worm -> Worm
;; Moves the worm in the direction it faces.
(defun worm-move (w)
  (make-worm
    (worm-dir w)
    (point+ (worm-head w)
      (dir-delta
        (worm-dir w)))
    (drop-last-point
      (cons (worm-head w)
        (worm-tail w))))
Dracula

;; short-worm : Integer Integer Direction -> Worm
;; Produces a worm without a tail.
(defun short-worm (x y d)
  (make-worm d (make-point x y) nil))

;; worm-turn : Worm Direction -> Worm
;; Changes the direction a worm faces.
(defun worm-turn (w d)
  (make-worm d (worm-head w) (worm-tail w)))

;; worm-move : Worm -> Worm
;; Moves the worm in the direction it faces.
(defun worm-move (the-worm)
  (make-worm
   (worm-dir the-worm)
   (point+ (worm-head the-worm)
   (dir-delta
    (worm-dir the-worm)))
   (drop-last-point
    (cons (worm-head the-worm)
      (worm-tail the-worm))))
Dracula

```lisp
;; short-worm : Integer Integer Direction -> Worm
;; Produces a worm without a tail.
(defun short-worm (x y d)
  (make-worm d (make-point x y) nil))

;; worm-turn : Worm Direction -> Worm
;; Changes the direction a worm faces.
(defun worm-turn (w d)
  (make-worm d (worm-head w) (worm-tail w)))

;; worm-move : Worm -> Worm
;; Moves the worm in the direction it faces.
(defun worm-move (w)
  (make-worm
    (worm-dir w)
    (point+ (worm-head w)
      (dir-delta
        (worm-dir w)))
    (drop-last-point
      (cons (worm-head w)
        (worm-tail w))))
```

Functional Programming and Theorem Proving for Undergraduates
FDPE 2008 - Rex Page / Carl Eastlund / Matthias Felleisen
Dracula

```lisp
;; sqr : Int -> Int
(defun sqr (x)
  (* x x))

;; All squares are nonnegative.
(defthm sqr>=0
  (implies (integerp x)
           (>= (sqr x) 0)))
```

Welcome to Clozure Common Lisp
Version 1.2-r10446M-RC1
(DarwinPPC64)!

ACL2 Version 3.4 built August 12, 2008 09:24:34.
Copyright (C) 2008
University of Texas at Austin
Dracula

;;; sqr : Int -> Int
(defun sqr (x)
  (* x x))

;;; All squares are nonnegative.
(defthm sqr>=0
  (implies (integerp x)
    (>= (sqr x) 0)))

Q.E.D.

Summary
Form:  ( DEFTHM SQR>=0 ...)
Rules: ( (:DEFINITION NOT)
          (:DEFINITION SQR))
;; sqr : Int -> Int  
(defun sqr (x)  
 (* x x))

;; All squares are nonnegative.
(defun sqr>=0  
 (implies (integerp x)  
    (>= (sqr x) 0)))
;; sqr : Int -> Int
(defun sqr (x)
  x)

;; All squares are nonnegative.
(defthm sqr>=0
  (implies (integerp x)
           (>= (sqr x) 0)))
Dracula

;;; sqr : Int -> Int
(defun sqr (x)
  x)

;;; All squares are nonnegative.
(defthm sqr>=0
  (implies (integerp x)
    (>= (sqr x) 0)))

(DEFTHM SQR>=0 ...)  
******** FAILED ********

(IMPLIES (INTEGERP X) (<= 0 X)).

Name the formula above *1.

No induction schemes are suggested by *1. Consequently, the proof attempt has failed.
Program Design

- How to Design Programs code:

```scheme
;;; sqr : Int -> Int
(define (sqr x)
  (* x x))

;;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
```
Program Design

- Dracula code:

```scheme
;;; sqr : Int -> Int
(defun sqr (x)
  (* x x))

;;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
```
Unit Tests

- Dracula code:

```scheme
;; sqr : Int -> Int
(defun sqr (x)
    (* x x))

;; Unit tests: (==> assert-event)
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
```
Unit Tests

```lisp
;;; sqr : Int -> Int
(defun sqr (x)
  x)

;;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
```

Test Results

Ran 2 checks.
1 of the 2 checks failed.

Actual value 2 differs from 4, the expected value.

In /Users/cce/Desktop/sqr.lisp at line 10 column 0

Functional Programming and Theorem Proving for Undergraduates
FDPE 2008 - Rex Page / Carl Eastlund / Matthias Felleisen
Unit Tests

```lisp
;;; sqr : Int -> Int
(defun sqr (x)
  (+ x x))

;;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
```

Welcome to DrScheme, version 4.1 [3m].
Language: Dracula v4.2.
All tests passed!
>
Beyond Unit Tests

;; sqr : Int -> Int
(defun sqr (x)
  (+ x x))

;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
Beyond Unit Tests

;; sqr : Int -> Int
(defun sqr (x) (+ x x))

;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)

;; All squares are nonnegative
(defthm sqr>=0
  (implies (integerp x) (>= 0 (+ x x))).

Name the formula above *1.

No induction schemes are suggested by *1. Consequently, the proof attempt has failed.
DoubleCheck

;;; ACL2 theorem:
(defthm name
  (implies (and precondition ...)
           postcondition))

;;; DoubleCheck property:
(defproperty name
  (x [:where precondition]
     [:value distribution] ...) postcondition)
DoubleCheck

;;; ACL2 theorem:
(defthm sqr>=0
  (implies (integerp x)
            (>= (sqr x) 0)))

;;; DoubleCheck property:
(defproperty sqr>=0
  (x)
  (implies (integerp x)
            (>= (sqr x) 0)))
DoubleCheck

;; ACL2 theorem:
(defthm sqr>=0
  (implies (integerp x)
    (>= (sqr x) 0)))

;; DoubleCheck property:
(defproperty sqr>=0
  (x :where (integerp x))
  (>= (sqr x) 0))
DoubleCheck

;; ACL2 theorem:
(defthm sqr>=0
  (implies (integerp x)
           (>= (sqr x) 0))

;; DoubleCheck property:
(defproperty sqr>=0
  (x :where (integerp x)
      :value (random-integer))
  (>= (sqr x) 0))
DoubleCheck

;;; Simple distributions:
(random-string)
(random-integer)

;;; Parameterized distributions:
(random-between low high)
(random-list-of dist [:size size])

;;; Write new distributions:
(defrandom name (arg ...) expr)
DoubleCheck

;;; ACL2 theorem:
(defthm sqr>=0
   (implies (integerp x)
            (>= (sqr x) 0)))

;;; DoubleCheck property: (==> defthm)
(defproperty sqr>=0
   (x :where (integerp x)
        :value (random-integer))
   (>= (sqr x) 0))
DoubleCheck

;;; ACL2 theorem:
(defthm sqr>=0
  (implies (integerp x)
    (>= (sqr x) 0)))

;;; Ideal syntax (future work):
(defproperty sqr>=0
  (implies (integerp x)
    (>= (sqr x) 0)))
DoubleCheck

```lisp
;;; sqr : Int -> Int
(defun sqr (x)
  (+ x x))

;;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)

;;; All squares are nonnegative
(defproperty sqr>=0
  (x :where (integerp x)
    :value (random-integer))
  (>= (sqr x) 0))
```

Additional information:
- `key x: `-30
- `key check-expect:
  (check-expect
    (let ((x '30))
      (>= (sqr x) 0)))
  `true`

Timing:
- `cpu: 0; real: 0; gc: 0`
DoubleCheck

;; sqr : Int -> Int
(defun sqr (x)
  (+ x x))

;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
(check-expect (let ((x '-30)) (> (sqr x) 0)) t)

Dracula v4.2

Ran 3 checks.
1 of the 3 checks failed.

Actual value \texttt{nil} differs from \texttt{t}, the expected value.

\texttt{In /Users/cce/Desktop/sqr.lisp at line 12 column 0}
DoubleCheck

;; sqr : Int -> Int
(defun sqr (x)
  (+ x x))

;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
(check-expect (sqr -30) 900)

Ran 3 checks.
1 of the 3 checks failed.

Actual value -60 differs from 900, the expected value.
In /Users/cce/Desktop/sqr.lisp at line 12 column 0

Functional Programming and Theorem Proving for Undergraduates
FDPE 2008 - Rex Page / Carl Eastlund / Matthias Felleisen
DoubleCheck

;; sqr : Int -> Int
(defun sqr (x)
  (+ x x))

;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
(check-expect (sqr -30) 900)
DoubleCheck

;;; sqr : Int -> Int
(defun sqr (x)
  (* x x))

;;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
(check-expect (sqr -30) 900)
DoubleCheck

;; sqr : Int -> Int
(defun sqr (x)
  (* x x))

;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
(check-expect (sqr -30) 900)

;; All squares are nonnegative.
(defproperty sqr>=0
  (x :where (integerp x)
      :value (random-integer))
  (>= (sqr x) 0))

Language: Dracula v4.2.
All tests passed!

DoubleCheck
Total: 50 successes
Successes (1/1)
sqr>=0

Q.E.D.

Summary
Form:  ( DEPTHM SQR>=0 ...)
Rules: ((:DEFINITION NOT)
         (:DEFINITION SQR)
Software Engineering Courses at OU

- **SE-i**
  - Process (30%) - Humphrey PSP
  - Design (35%) - FP in ACL2
  - Testing/Validation (35%)
    - Predicate-based, automated testing (DblChk)
    - Mechanized logic for full verification (ACL2)
  - Software development projects
    - 6 individual projects: Design/Code/PSP rpt
      - Early projects: small components
      - Later projects: applications using components
    - 2 team projects
      - Building on components and applications
      - Seven deliverables in all
Software Engineering Courses at OU

- **SE-i**
  - **Process (30%)** - Humphrey PSP
  - **Design (35%)** - FP in ACL2
  - **Testing/Validation (35%)**
    - Predicate-based, automated testing (DblChk)
    - Mechanized logic for full verification (ACL2)
  - **Software development projects**
    - 6 individual projects: Design/Code/PSP rpt
      - Early projects: small components
      - Later projects: applications using components
    - 2 team projects
      - Building on components and applications
      - Seven deliverables in all

60% 30% 10% other
Software Engineering Courses at OU

- **SE-ii**
  - Organized around one sfw devp project
  - Team project (4 - 6 students per team)
  - Project size
    - 3,000 - 5,000 lines of code, before ACL2
    - 2,000 - 3,000 lines of code, since intro of ACL2
  - 12 separate (team) deliverables
    - Engineering std, design/schedule, code, installation/usage doc, defect history, tests/theorems, meeting log, ...
    - 3 presentations - last to Advisory Board
  - Individual journals — expanded PSP rpt
Background of SE Students

- **Standard CS curriculum**
  - ABET, math heavy
- **No significant FP experience**
  - Minor exposure in PL course
- **Serious logic course (70% of students)**
  - Reasoning about hdw/sfw properties
- **So, SE is first serious exposure to FP**
  - Almost all succeed in
    - Learning FP
    - Predicate-based testing
- **Success with ACL2 mechanized logic**
  - Most acquire a reasonable level of comfort
  - 10% to 20% gain proficiency with ACL2 logic
Example SE-i Project

- **Linear encode/decode**
  - Message: \( x_0 \ x_1 \ldots \ x_{n-2} \ x_{n-1} , \ 0 \leq x_k < m \)
  - Encoding: \( \ldots (x_k + x_{k+1}) \mod m \ldots \), where \( x_n = m - 1 \)

- **Define encode, decode, and predicates**
  - encode, decode, code-list?

- **Define correctness properties**
  - \( k^{\text{th}} \) element of encoded list is \( (x_k + x_{k+1}) \mod m \)
  - decode inverts encode
Example SE-i Project

- Linear encode/decode
  - Message: \( x_0 \ x_1 \ \ldots \ x_{n-2} \ x_{n-1} , \ 0 \leq x_k < m \)
  - Encoding: \( \ldots (x_k + x_{k+1}) \mod m \ldots \), where \( x_n = m-1 \)

- Define correctness properties
  - decode inverts encode

- Inversion property
  
  (defproperty decode-inverts-encode
    (m :value (random-between 2 100)
    (xs :value (random-list-of
              (random-between 0 (- m 1))))
    (equal (decode m (encode m xs)) xs)))
Example SE-i Project

- Linear encode/decode
  - Message: $x_0 \ x_1 \ \ldots \ x_{n-2} \ x_{n-1}$, $0 \leq x_k < m$
  - Encoding: $\ldots (x_k + x_{k+1}) \mod m \ldots$, where $x_n = m-1$

- Define correctness properties
  - $\text{decode}$ inverts $\text{encode}$

- Inversion property as (untrue) theorem
  (defthm decode-inverts-encode-thm
   (equal (decode m (encode m xs)) xs)))
Example SE-i Project

- Linear encode/decode
  - Message: \( x_0 \ x_1 \ldots \ x_{n-2} \ x_{n-1} \), \( 0 \leq x_k < m \)
  - Encoding: \( \ldots (x_k + x_{k+1}) \mod m \ldots \), where \( x_n = m-1 \)

- Define correctness properties
  - decode inverts encode

- Inversion property with preconditions

  ```lisp
  (defproperty decode-inverts-encode
    (m :where (and (integerp m) (>= m 2))
    :value (random-between 2 100)
    xs :where (code-list? m xs)
    :value (random-list-of
             (random-between 0 (- m 1))))
  (equal (decode m (encode m xs)) xs)))
  ```
Example SE-i Project

- Linear encode/decode
  - Message: \( x_0 \ x_1 \ldots \ x_{n-2} \ x_{n-1} , \ 0 \leq x_k < m \)
  - Encoding: \( \ldots (x_k + x_{k+1}) \mod m \ldots \), where \( x_n = m-1 \)

- Define correctness properties
  - decode inverts encode

- Inversion property as theorem
  \[
  \text{(defthm decode-inverts-encode-thm)}
  \qquad \text{(implies (and (integerp m)}
  \qquad \qquad (>= m 2)
  \qquad \qquad (code-list? m xs))
  \qquad \text{(equal (decode m (encode m xs))}
  \qquad \text{x\(s\)))}
  \]
Example SE-i Project

- Linear encode/decode
  - Message: \( x_0 \ x_1 \ \ldots \ x_{n-2} \ x_{n-1} \), \( 0 \leq x_k < m \)
  - Encoding: \( \ldots (x_k + x_{k+1}) \mod m \ldots \), where \( x_n = m-1 \)

- Define correctness properties
  - \( k^{th} \) element of encoded list is \( (x_k + x_{k+1}) \mod m \)

- Right-stuff property as (untrue) theorem

```lisp
(defthm encoded-elements-are-correct-thm
  (implies (and (integerp m) (>= m 2)
               (code-list? m xs)
               (integerp k))
           (= (nth k (encode m xs))
               (mod (+ (nth k xs) (nth (+ k 1) xs)) m))))
```

Functional Programming and Theorem Proving for Undergraduates
FDPE 2008 - Rex Page / Carl Eastlund / Matthias Felleisen
Example SE-i Project

- Linear encode/decode
  - Message: $x_0 \ x_1 \ldots \ x_{n-2} \ x_{n-1}$, $0 \leq x_k < m$
  - Encoding: $\ldots (x_k + x_{k+1}) \mod m \ldots$, where $x_n = m-1$

- Define correctness properties
  - $k^{th}$ element of encoded list is $(x_k + x_{k+1}) \mod m$

- Right-stuff property as theorem
  (defthm encoded-elements-are-correct-thm
   (implies (and (integerp m) (>= m 2) (code-list? m xs) (natp k) (< k (- (len xs) 1)) (= (nth k (encode m xs)) (mod (+ (nth k xs) (nth (+ k 1) xs)) m))))

Functional Programming and Theorem Proving for Undergraduates
FDPE 2008 - Rex Page / Carl Eastlund / Matthias Felleisen
Example SE-i Project

- Linear encode/decode
  - Message: \( x_0 \ x_1 \ \ldots \ x_{n-2} \ x_{n-1} \), \( 0 \leq x_k < m \)
  - Encoding: \( \ldots (x_k + x_{k+1}) \mod m \ldots \), where \( x_n = m-1 \)
- Define correctness properties
  - \( k\)th element of encoded list is \( (x_k + x_{k+1}) \mod m \)
- **Right-stuff property as vacuous theorem**

```lisp
(defun encoded-elements-are-correct-thm
  (implies (and (integerp m) (>= m 2) (code-list? m xs) (<= k 0) (> k (len xs)))
            (= (nth k (encode m xs))
                (mod (+ (nth k xs) (nth (+ k 1) xs)) m)))
```

Functional Programming and Theorem Proving for Undergraduates
FDPE 2008 - Rex Page / Carl Eastlund / Matthias Felleisen
Team Project Example from SE-ii

- Conway game of life (cellular automaton)
  - Multiple topologies - sphere, cylinder, torus, Klein
  - Six solutions, 1200 - 7000 lines of code, avg: 3000
  - 7000-line implementation included
    - Three-dimensional rendering
    - Over 100 properties verified by ACL2 mechanized logic
    - Ten properties on 3D-rendering (eg, no bit-plane errors)

![Gosper glider gun](image)
Reactions to SE Courses

- **Students**
  - PSP unpopular (time & defect logs, plans...)
  - Functional programming
    - Almost all get it, eventually
    - 10% complain
    - 10% - 20% really like it
    - The rest take it as an interesting challenge
  - Property-based testing
    - Just started this semester
    - Students seem to like it
    - Smoothes the way towards theorems
- **Theorems**
  - Top quarter like it, bottom quarter gets lost
- **Advisory board (from computing industry)**
  - Positive comments nearly universal
Outreach

- Three-day workshop, May 2008
  - Participants: 13 CS instructors from 6 states
  - Lectures (35%) plus hands-on projects (65%)
  - Two leaders, plus two aids with ACL2 expertise

- Lessons learned
  - Theorems are easier than automated testing
    - Appropriate random distributions add complication
  - Specifying properties requires careful thought
    - Incorrect or vacuous theorems—common first attempts
    - Payoff—better understanding of software
  - Projects must be carefully constructed
    - Ensure reasonable solutions (solve them in advance)

- MEPLS semiannual meeting
Outreach

- Three-day workshop, May 2008
  - Participants: 13 CS instructors from 6 states
  - Lectures (35%) plus hands-on projects (65%)
  - Two leaders, plus two aids with ACL2 expertise

- Lessons learned
  - Theorems are easier than automated testing
    - Appropriate random distributions add complication
  - Specifying properties requires careful thought
    - Incorrect or vacuous theorems—common first attempts
    - Payoff—better understanding of software
  - Projects must be carefully constructed
    - Ensure reasonable solutions (solve them in advance)

- MEPLS semiannual meeting

- **Google** Dracula DrScheme, Rex SEcollab, MEPLS
Plans for Future

- Integrated testing / verification
- Dracula module facility
- Coordinated projects (on website)
  - Building from components to applications
  - Four tracks, 4 - 6 projects in each track
- Outreach workshops
  - SIGCSE tutorial
  - Three-day workshops
The End