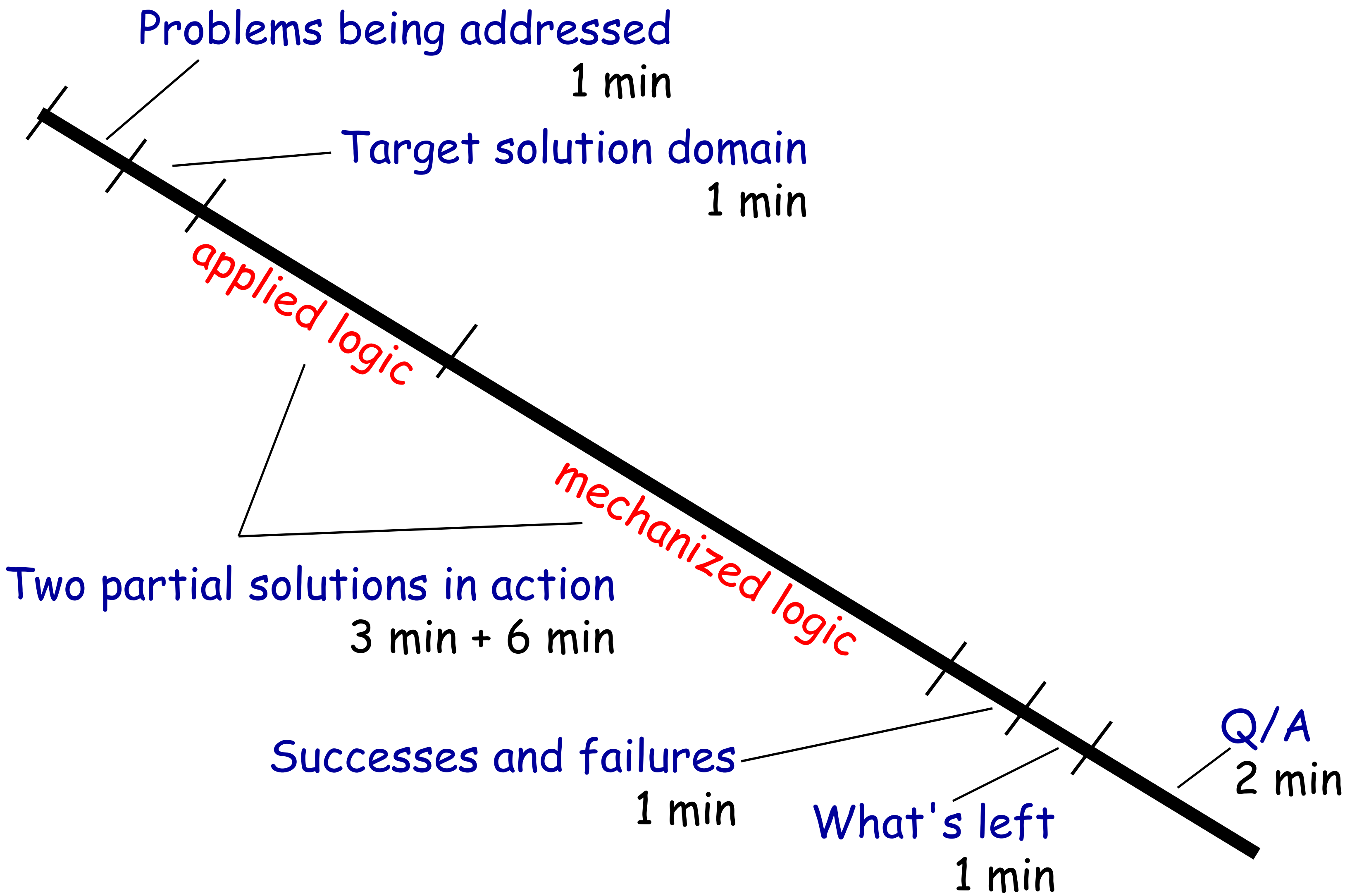
Computational Logic in the Undergraduate Curriculum

Rex Page
University of Oklahoma

ACL2 Workshop 8
Boston — May 11, 2009

NSF DUE-0633004
Integrating Mechanized Logic into the SE Curriculum
Collaborating with Matthias Felleisen, Northeastern University
NSF EIA-0082849
Formal Methods Education and Programming Effectiveness

Timeline – 15 minute presentation



The Problem

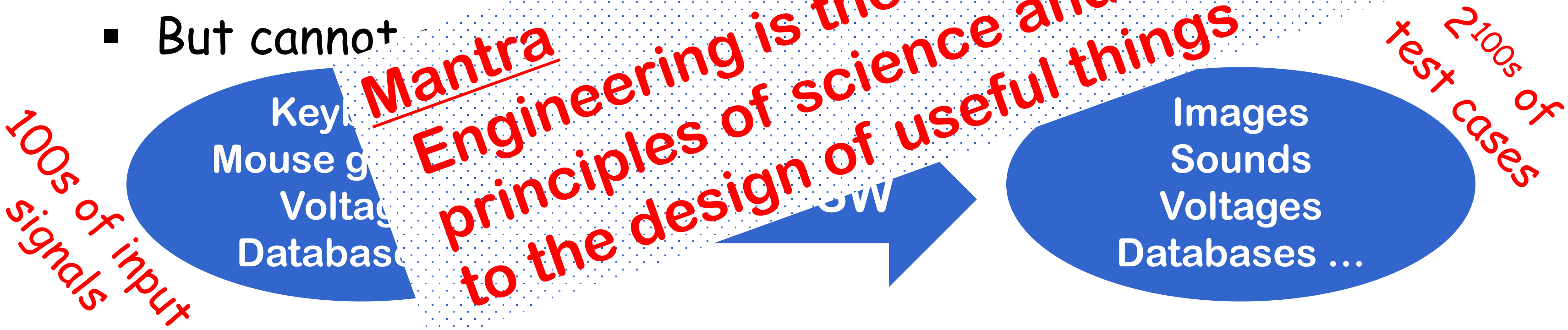
- Software is full of bugs
  - Hardware, too – fewer, but expensive to fix
- Testing helps
  - But cannot guarantee properties

- Software and hardware = formulas in mathematical logic
  - Complicated formulas, but formulas, nevertheless
  - So, they are amenable to mathematical analysis
- Applying mathematical logic to hw/sw
  - Offers possibility of fully verified hw/sw properties
  - Questions
    - ✓ How much does it cost?
    - ✓ How do we find engineers who can apply formal methods?

Education plays a key role

The Problem

- Software is full of bugs
  - Hardware, too – fewer, but exp
- Testing helps
  - But cannot

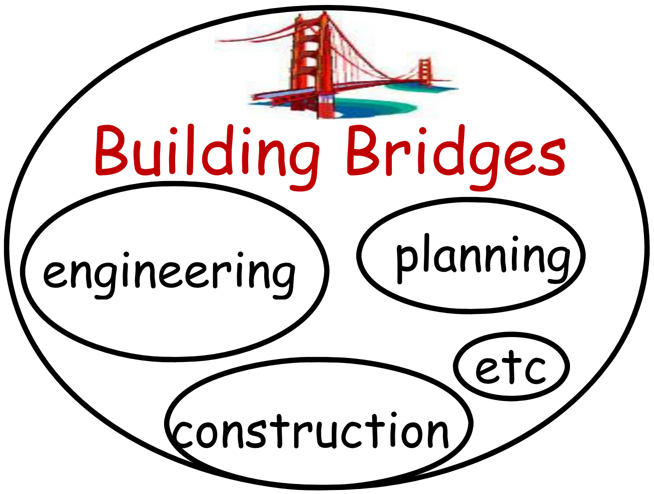


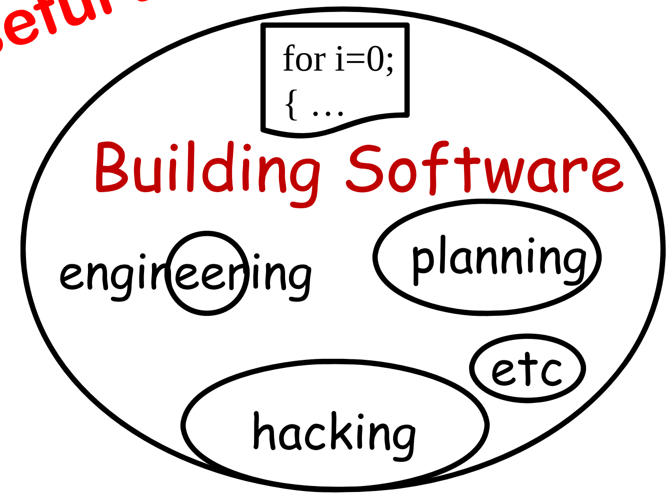
- Software and hardware = formulas in mathematical logic
  - Complicated formulas, but formulas, nevertheless
  - So, they are amenable to mathematical analysis
- Formal methods = applying mathematical logic to hw/sw
  - Offers possibility of fully verified hw/sw properties
  - Questions
    - ✓ How much does it cost?
    - ✓ How do we find engineers who can apply formal methods?

Education plays a key role

Engineering Disciplines

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





SE2004 Curriculum

http://sites.computer.org/ccse/

SE2004 has little if any material on what the dictionary calls "engineering"

Opportunities for Logic in CS Curriculum

- Courses required in 90% of CS programs
  - Programming I/II
  - Data structures
  - Operating systems
  - Computer organization
  - Discrete math
  - Software engineering

random survey

No other course required in over 60% of curricula

- Undergrad CS courses at University of Oklahoma

Required

yr 1
- ~~Intro to Computer Programming~~
- ~~Prog'g Structures & Abstractions~~

yr 2
- ~~Data Structures~~
- Applied Logic for Hdw and Sfw
- Computer Organization
- Discrete Math

yr 3
- Theory of Computation
- ~~Human Computer Interaction~~
- ~~Intro to Operating Systems~~
- Principles of Prog'g Languages

yr 4
- ~~Numerical Methods~~
- Software Engineering I
- Software Engineering II
- Algorithm Analysis

Elective (choose 3) — 4th year
- Artificial Intelligence
- Intro to Intelligent Robotics
- Machine Learning
- Compiler Construction
- ~~Computer Graphics~~
- Database Management
- ~~Computer Architecture~~
- Data Networks
- Embedded Systems
- Operating Systems Theory
- ~~Scientific Computing~~
- ~~Discrete Optimization~~
- Cryptography

Applying Principles of Math in CS Courses

- **Discrete Mathematics**. Mathematical foundations of CS.
  Topics: combinatorics, graph theory, relations,
  functions, logic,
  computational complexity, recurrences
  - Traditional course: 50% blue, 25% red, 25% green
  - Formal methods-based course: 20% blue, 60% red, 20% green

http://www.cs.ou.edu/~beseme/sfwIsDMpaper.pdf

- **Software Engineering**
  - Traditional course
    - 60% process, 20% design, 20% testing
    - Conventional programming (C++, Java, …)
  - Formal methods-based course
    - 30% process, 35% design, 35% testing/verification
    - Equation-based programming and mechanized logic (ACL2)
    - Programming environment – DrScheme/DrACuLa
    - Why this particular technology?
      - ✓ Theorem-proving scales to large systems
      - ✓ ACL2 combines programming language/mechanized logic
      - ✓ Learning curve is not steep

http://www.cs.ou.edu/~rlpage/SEcollab/EngrSwJFP.pdf

Logic Applied to Sw/Hw

- Logic part of Discrete Math course
  - Natural deduction (Goentzen) —2 weeks (four 75-min classes)
  - Boolean algebra — 1 week
  - Predicates — 1 week
  - Induction —5 weeks
    - ✓ Reasoning about software
- Applied Logic Course
  - All that + circuits
  - Reasoning about circuits — 5 weeks

- Artifacts analyzed (properties proved using formal logic)

Software Components
sum
logic operations (and, or, …)
list operations (len, concat, fold, …)
maximum
merge-sort
quicksort
binary numerals
AVL insertion

Hardware Components
circuit minimization (Karnaugh)
half-adder, full-adder
ripple-carry adder
sum of list (sequential circuit)

- Mostly correctness properties
- A few performance properties

Example: Properties of Merge-Sort

```
dmx(x : y : xys) = (x : ys, y : ys)                       --dmx::
  where (xs, ys) = dmx xs
dmx[x] = ([x],[ ])                                         --dmx[x]
dmx[ ] = ([ ],[ ])                                         --dmx[ ]
merge (x : xs) (y : ys) =
  if y < x then y :(merge (x : xs) ys)                     --merge::y
           else x :(merge xs (y : ys))                     --merge::x
merge (x : xs) [ ] = (x : xs)                              --merge:
merge [ ] ys = ys                                          --merge[ ]
msort(x1 : x2 : xs) = merge (msort ys) (msort zs)          --msort::
  where (ys, zs) = dmx(x1 : x2 : xs)
msort[x] = [x]                                             --msort[x]
msort[ ] = [ ]                                             --msort[ ]
```

axioms

preservation of length

length(msort xs) = (length xs) {msL}

length(merge xs ys) = (length xs) + (length ys) {mL}

(ys, zs)=(dmx xs) $\rightarrow$ (length ys) + (length zs) = (length xs) {dL}

(length xs)>1) $\wedge$ (ys, zs)=(dmx xs) $\rightarrow$ (length ys)<(length xs) $\wedge$ (length zs)<(length xs) {dL<}

preservation of list elements

(elem x xs) $\rightarrow$ (elem x (msort xs)) {msE}

((elem x xs) $\vee$ (elem x ys)) $\rightarrow$ (elem x (merge xs ys)) {mE}

((elem x xs) $\wedge$ (ys, zs)=dmx xs) $\rightarrow$ ((elem x ys) $\vee$ (elem x zs)) {dE}

theorems

Merge-Sort Preserves Length

```
msort(x₁ : x₂ : xs) = merge (msort ys) (msort zs)      --msort::
  where (ys,zs) = dmx(x₁ : x₂ : xs)
msort[x] = [x]                                          --msort[x]
msort[ ] = [ ]                                          --msort[ ]
```

- Theorem msL: length(msort xs) = (length xs) {P(length xs)}

Proof:

P(0)

length(msort[ ])
= length[ ] {msort[]}
= 0 {len[]}

P(1)

length(msort[x])
= length[x] {msort[x]}
= 1 {(:),len:,len[],+}

P(n+2)

length(msort(x_1 : x_2 : xs)) {(:), (:)}
= length(merge (msort ys) (msort zs)) {msort::}
where (ys, zs) = dmx(x_1 : x_2 : xs)
= length(msort ys) + length(msort zs) {mL}
= (length ys) + (length zs) {dL<,P(length ys),P(length zs)}
= length xs {dL}

QED(msL) by induction on (length xs)

Software Engineering Course

- ❑ Next part of this talk
  - Some typical lecture material from the SE course
  - Pretty much as done in the course, but . . .
  - Faster pace, plus a few short cuts
    - ✓ 5 min, instead of 150 minutes
- ❑ This material could be a one-week module
  - Three 50-minute lectures
  - Lecture 1: Lisp basics
    - ✓ Data structures: atoms, lists
    - ✓ Operators: car, cdr, cons, if, equal
    - ✓ Function definitions: defun
  - Lectures 2, 3: Testing, logic, verification
    - ✓ Test-driven development
    - ✓ Predicate-based testing
    - ✓ Verification of properties using theorem prover

Structural Induction

- Define demultiplexor using structural induction on lists
  - (dmx (x_1 y_1 x_2 y_2 x_3 y_3 …)) = ((x_1 x_2 x_3 …) (y_1 y_2 y_3 …))

template

```
(defun dmx (xys)
  (if (test-for-non-inductive-case xys)
      (g xys)
      (h (car xys) (dmx (cdr xys)))))
```

Fill in the missing red parts

- Design
  - Suppose xys = (x_1 y_1 x_2 y_2 x_3 y_3 …)
  - Then, (cdr xys) = (y_1 x_2 y_2 x_3 y_3 x_4 …)
  - So, (dmx (cdr xys)) = ((y_1 y_2 y_3 …) (x_2 x_3 x_4 …))
  - Now, work out test, g, and h and paste the whole thing together

```
(defun h (x ysxs)  ; x=x1 ysxs=((y1 y2 y3 ...) (x2 x3 x4 ...)) )
  (list (cons x (cadr ysxs)) (car ysxs)))
(defun dmx (xys)
  (if (endp xys)
      (list nil nil)
      (h (car xys) (dmx (cdr xys)))))
```

A Derived Property of dmx

(dmx (x_0 y_0 x_1 y_1 x_2 y_2 …)) = ((x_0 x_1 x_2 …) (y_0 y_1 y_2 …))

- Length preserved (predicate-based test: DoubleCheck)

```
(defproperty dmx-preserves-length-tst :repeat 100
  (xys :value (random-list-of (random-integer)))
  (implies (true-listp xys)
           (= (+ (len (car  (dmx xys)))
                 (len (cadr (dmx xys))))
              (len xys))))
```

A Derived Property of dmx

(dmx (x_0 y_0 x_1 y_1 x_2 y_2 …)) = ((x_0 x_1 x_2 …) (y_0 y_1 y_2 …))

- Length preserved (predicate-based test: DoubleCheck)

```
(defproperty dmx-preserves-length-tst :repeat 100
  (xys :value (random-list-of (random-integer)))
  (implies (true-listp xys)
           (= (+ (len (car  (dmx xys)))
                 (len (cadr (dmx xys))))
              (len xys))))
```

- Here is the theorem Dracula derives from above property

```
(defthm dmx-preserves-length
  (implies (true-listp xys)
        (= (+ (len (car  (dmx xys)))
              (len (cadr (dmx xys))))
           (len xys))))
```

dmx: Conservation of Elements
another derived property

$(\text{dmx } (x_0\ y_0\ x_1\ y_1\ x_2\ y_2\ \ldots)) = ((x_0\ x_1\ x_2\ \ldots)\ (y_0\ y_1\ y_2\ \ldots))$

- dmx does not lose (or gain) list elements

```
(defproperty dmx-conservation-of-elements-tst :repeat 100
 (xys :value (random-list-of (random-between 0 10))
  e   :value (random-between 0 20))
 (implies (true-listp xys)
          (iff (member-equal e xys)
               (or (member-equal e (car  (dmx xys)))
                   (member-equal e (cadr (dmx xys)))))))
```

- Corresponding theorem

```
(defthm dmx-conservation-of-elements
 (implies (true-listp xys)
          (iff (member-equal e xys)
               (or (member-equal e (car  (dmx xys)))
                   (member-equal e (cadr (dmx xys)))))))
```

We have defined a demultiplexor
How about the multiplexor?

- Demultiplexor
  - (dmx (x_0 y_0 x_1 y_1 x_2 y_2 …)) = ((x_0 x_1 x_2 …) (y_0 y_1 y_2 …))
- Multiplexor
  - (mux (x_0 x_1 x_2 …) (y_0 y_1 y_2 …)) = (x_0 y_0 x_1 y_1 x_2 y_2 …)

structural induction with two operands

```
(defun mux (xs ys)
  (if (test-for-non-inductive-case xs ys)
      (g xs ys)
      (h (car xs) (car ys) (mux (cdr xs) (cdr ys)))))
```

```
(defun mux (xs ys)
  (if (endp xs)
      ys
      (if (endp ys)
          xs
          (append (list (car xs) (car ys))
                  (mux (cdr xs) (cdr ys))))))
```

Length Property of Multiplexor

(mux (x_1 x_2 x_3 …) (y_1 y_2 y_3 …)) = (x_1 y_1 x_2 y_2 x_3 y_3 …)

- Length preservation for dmx discussed before

```
(defproperty dmx-preserves-length-tst :repeat 100
  (xys :value (random-list-of (random-natural)))
  (implies (true-listp xys)
           (= (+ (len (car  (dmx xys)))
                 (len (cadr (dmx xys))))
              (len xys))))
```

- Similar property for mux

```
(defproperty mux-preserves-length-tst :repeat 100
  . . . specify values . . .
  . . . define length property . . . )
```

Length Property of Multiplexor

(mux (x_1 x_2 x_3 …) (y_1 y_2 y_3 …)) = (x_1 y_1 x_2 y_2 x_3 y_3 …)

- Length preservation for dmx discussed before

```
(defproperty dmx-preserves-length-tst :repeat 100
  (xys :value (random-list-of (random-natural)))
  (implies (true-listp xys)
           (= (+ (len (car  (dmx xys)))
                 (len (cadr (dmx xys))))
              (len xys))))
```

- Similar property for mux

```
(defproperty mux-preserves-length-tst :repeat 100
  (xs :value (random-list-of (random-natural))
   ys :value (random-list-of (random-natural)))
  (implies (and (true-listp xs) (true-listp ys))
           (= (+ (len xs) (len ys))
              (len (mux xs ys)))))
```

Multiplexor Law of Conservation

(mux (x_1 x_2 x_3 …) (y_1 y_2 y_3 …)) = (x_1 y_1 x_2 y_2 x_3 y_3 …)

- Conservation law for dmx

```
(defproperty dmx-conservation-of-elements :repeat 100
 (xys :value (random-list-of (random-between 0 10))
  e :value (random-between 0 20))
 (implies (true-listp xys)
          (iff (member-equal e xys)
               (or (member-equal e (car  (dmx xys)))
                   (member-equal e (cadr (dmx xys)))))))
```

- Similar property for mux

```
 (defproperty mux-conservation-of-elements :repeat 100
    . . . specify values . . .
    . . . define conservation property . . .  )
```

Multiplexor Law of Conservation

$(\text{mux } (x_1\ x_2\ x_3\ \ldots)\ (y_1\ y_2\ y_3\ \ldots)) = (x_1\ y_1\ x_2\ y_2\ x_3\ y_3\ \ldots)$

- Conservation law for dmx

```
(defproperty dmx-conservation-of-elements :repeat 100
 (xys :value (random-list-of (random-between 0 10))
  e :value (random-between 0 20))
 (implies (true-listp xys)
          (iff (member-equal e xys)
               (or (member-equal e (car  (dmx xys)))
                   (member-equal e (cadr (dmx xys)))))))
```

- Similar property for mux

```
(defproperty mux-conservation-of-elements :repeat 100
  (xs :value (random-list-of (random-between 0 10))
   ys :value (random-list-of (random-between 0 10))
   e  :value (random-between 0 20))
  (implies (and (true-listp xs) (true-listp ys))
           (iff (member-equal e (mux xs ys))
                (or (member-equal e xs)
                    (member-equal e ys)))))
```

mux and dmx invert each other

(mux (x_1 x_2 x_3 …) (y_1 y_2 y_3 …)) = (x_1 y_1 x_2 y_2 x_3 y_3 …)
(dmx (x_1 y_1 x_2 y_2 x_3 y_3 …)) = ((x_1 x_2 x_3 …) (y_1 y_2 y_3 …))

- mux inverts dmx

```
(defproperty mux-inverts-dmx-tst :repeat 100
 . . . specify values . . .
 . . . define inversion property . . . )
```

mux and dmx invert each other

$(\text{mux } (x_1\ x_2\ x_3\ \ldots)\ (y_1\ y_2\ y_3\ \ldots)) = (x_1\ y_1\ x_2\ y_2\ x_3\ y_3\ \ldots)$

$(\text{dmx } (x_1\ y_1\ x_2\ y_2\ x_3\ y_3\ \ldots)) = ((x_1\ x_2\ x_3\ \ldots)\ (y_1\ y_2\ y_3\ \ldots))$

- mux inverts dmx

```
(defproperty mux-inverts-dmx-tst :repeat 100
  (xys :value (random-list-of (random-natural)))
  (implies (true-listp xys)
           (equal (mux (car (dmx xys))
                       (cadr (dmx xys)))
                  xys)))
```

mux and dmx invert each other

(mux (x_1 x_2 x_3 …) (y_1 y_2 y_3 …)) = (x_1 y_1 x_2 y_2 x_3 y_3 …)
(dmx (x_1 y_1 x_2 y_2 x_3 y_3 …)) = ((x_1 x_2 x_3 …) (y_1 y_2 y_3 …))

- mux inverts dmx

```
(defproperty mux-inverts-dmx-tst :repeat 100
  (xys :value (random-list-of (random-natural)))
  (implies (true-listp xys)
           (equal (mux (car (dmx xys))
                       (cadr (dmx xys)))
                  xys)))
```

- dmx inverts mux

```
(defproperty dmx-inverts-mux-tst :repeat 100
  (xs :value (random-list-of (random-natural))
   ys :value (random-list-of (random-natural)))
  (implies (and (true-listp xs) (true-listp ys))
           (equal (dmx (mux xs ys))
                  (list xs ys))))
```

Not quite … Constrain to equal lengths

mux and dmx invert each other

$(\text{mux } (x_1\ x_2\ x_3\ \ldots)\ (y_1\ y_2\ y_3\ \ldots)) = (x_1\ y_1\ x_2\ y_2\ x_3\ y_3\ \ldots)$

$(\text{dmx } (x_1\ y_1\ x_2\ y_2\ x_3\ y_3\ \ldots)) = ((x_1\ x_2\ x_3\ \ldots)\ (y_1\ y_2\ y_3\ \ldots))$

- mux inverts dmx

```
(defproperty mux-inverts-dmx-tst :repeat 100
  (xys :value (random-list-of (random-natural)))
  (implies (true-listp xys)
           (equal (mux (car (dmx xys))
                       (cadr (dmx xys)))
                  xys)))
```

- dmx inverts mux

```
(defproperty dmx-inverts-mux-tst :repeat 100
  (n  :value (random-data-size)
   xs :value (random-list-of(random-natural) :size n)
   ys :value (random-list-of(random-natural) :size n))
  (implies (and (true-listp xs) (true-listp ys)
                (= (len xs) (len ys)))
           (equal (dmx (mux xs ys))
                  (list xs ys))))
```

dmx gets the right elements

(dmx (x_0 y_0 x_1 y_1 x_2 y_2 …)) = ((x_0 x_1 x_2 …) (y_0 y_1 y_2 …))

- Suppose every-other takes every other element
  - (every-other (x_0 x_1 x_2 x_3 x_4 …)) = (x_0 x_2 x_4 …)
  - Relationship between dmx and every-other

```
(defproperty dmx-evens=xs-tst :repeat 100
  (xys :value (random-list-of (random-natural)))
  (implies (true-listp xys)
           (equal (car (dmx xys))
                  (every-other xys))))
```

- Suppose (every-odd (x_0 x_1 x_2 x_3 x_4 …)) = (x_1 x_3 x_5 …)
  - Relationship between dmx and every-odd

```
(defproperty dmx-odds=ys-tst :repeat 100
  (xys :value (random-list-of (random-natural)))
  (implies (true-listp xys)
           (equal (cadr (dmx xys))
                  (every-odd xys))))
```

- Define every-other with structural induction template

Multiplexor Puts Elements in Right Places

(mux (x_0 x_1 x_2 …) (y_0 y_1 y_2 …)) = (x_0 y_0 x_1 y_1 x_2 y_2 …)

- dmx gets the right elements

```
(defproperty dmx-evens=xs-tst :repeat 100
  (xys :value (random-list-of (random-natural)))
  (implies (true-listp xys)
           (equal (car (dmx xys))
                  (every-other xys))))
```

- Similar property for mux

```
(defproperty mux-evens=xs-tst :repeat 100
  (xs :value (random-list-of (random-natural))
   ys :value (random-list-of (random-n          )
  (implies (and (true-listp                      ))
           (equal (
(defproperty                tst :repeat 100
  (xs :value (random-list-of (random-natural))
   ys :value (random-list-of (random-natural)))
  (implies (and (true-listp xs) (true-listp ys))
           (equal (every-odd (mux xs ys))
                  ys)))
```

Yikes! What's wrong?
Maybe we'd better look at results

Multiplexor Puts Elements in Right Places

(mux (x_0 x_1 x_2 …) (y_0 y_1 y_2 …)) = (x_0 y_0 x_1 y_1 x_2 y_2 …)

- mux puts elements in the right places

```
(defproperty mux-evens=xs-tst :repeat 100
  (n  :value (random-data-size)
   xs :value (random-list-of (random-natural) :size n)
   ys :value (random-list-of (random-natural) :size n))
  (implies (and (true-listp xs) (true-listp ys)
                (= (len xs) (len ys)))
           (equal (every-other (mux xs ys))
                  xs)))
(defproperty mux-odds=ys-tst :repeat 100
 (n  :value (random-data-size)
   xs :value (random-list-of (random-natural) :size n)
   ys :value (random-list-of (random-natural) :size n))
  (implies (and (true-listp xs) (true-listp ys)
                (= (len xs) (len ys)))
           (equal (every-odd (mux xs ys))
                  ys)))
```

Will Students Accept This Approach?

- In my experience, yes
  - All learn to develop programs and use DoubleCheck
  - Almost all have a positive experiences with at least some theorems
  - 5% - 10% acquire some proficiency with the theorem prover
- Why don't they rebel?
  - One of the few things they learn in programming that they didn't know in high school (or before)
- Industry advisors of CS department say they like it
  - Emphasis on correctness and disciplined testing
- Developing courses is a LOT OF WORK … Who will do it?
  - It's not easy to get tenure/promotion credit for this stuff
  - Might help if goods materials were available on the web
  - It's not "if you build it, they will come", though
- Pitfalls to avoid
  - Courses that the faculty micromanages
  - Untested projects
    - ✓ Make sure ACL2 can prove some theorems (correctly stated)
    - ✓ A few can require lemmas (challenges good students)

Questions?