# Uses of proof assistants

- Traditional use of proof assistants is to prove theorems
  - major application is formal verification (FV)
  - hardware, software, protocols etc
- But can also use deduction when main goal is not a theorem
  - refinement (check preconditions of rules)
  - symbolic simulation (`computeLib`, `Acl2`, `PVS`)
  - synthesis/compilation by rewriting
  - coverage analysis – locating unreachable states/transitions
- PROSPER project embedded a 'proof engine' (Hol98) in
  - smart Excel spreadsheet
  - the IFAD VDM Toolbox
- Embedded proof may be an important future role of automated deduction
  - HOL-4: nice environment for building applications with '*Proof Inside*'

# PuzzleTool

- Illustrates building a special purpose tool
  - meant to be analogous to ‘point tools’ in ECAD
- Combination of theorem proving and BDD calculation
  - use ‘LCF approach’ to using BDDs from TPHOLs 2000 paper
- Solves a class of puzzles generalising ‘peg solitaire’
  - a ‘point tool’ for puzzle designers!

## Flow from problem to solution

- Input problem represented naturally in logic
  - as sets of integers
- Derive by proof a representation suitable for symbolic calculation
  - one boolean variable for every potential set element
- Invoke calculation tools
  - `HolBddLib`
- Translate solution into problem specification representation
  - abstract from formulas about booleans to sets of integers

# Peg Solitaire

- Played on a board with holes into which pegs are inserted/removed

```
                    -------------------------
                    | XXXXX | XXXXX | XXXXX |
                    -------------------------
                    | XXXXX | XXXXX | XXXXX |
    ---------------------------------------------------------
    | XXXXX | XXXXX | XXXXX | XXXXX | XXXXX | XXXXX | XXXXX |
    ---------------------------------------------------------
    | XXXXX | XXXXX | XXXXX |       | XXXXX | XXXXX | XXXXX |
    ---------------------------------------------------------
    | XXXXX | XXXXX | XXXXX | XXXXX | XXXXX | XXXXX | XXXXX |
    ---------------------------------------------------------
                    | XXXXX | XXXXX | XXXXX |
                    -------------------------
                    | XXXXX | XXXXX | XXXXX |
                    -------------------------
```

- Each move removes one peg
- Goal is to end up with one peg in centre
- I got the example from Bill Roscoe, who used FDR to solve it

# BDD approach to Peg Solitaire

- Standard approach: a boolean variable $\mathtt{v}n$ for each of the 33 positions

```
                    -------------------------
                    |  v00  |  v02  |  v03  |
                    -------------------------
                    |  v04  |  v05  |  v06  |
    ---------------------------------------------------------
    |  v07  |  v08  |  v09  |  v10  |  v11  |  v12  |  v13  |
    ---------------------------------------------------------
    |  v14  |  v15  |  v16  |  v17  |  v18  |  v19  |  v20  |
    ---------------------------------------------------------
    |  v21  |  v22  |  v23  |  v24  |  v25  |  v26  |  v27  |
    ---------------------------------------------------------
                    |  v28  |  v29  |  v30  |
                    -------------------------
                    |  v31  |  v32  |  v33  |
                    -------------------------
```

- Initial state: $\mathtt{v1} \wedge \mathtt{v2} \wedge \cdots \wedge \mathtt{v16} \wedge \neg\mathtt{v17} \wedge \mathtt{v18} \wedge \cdots \wedge \mathtt{v33}$
- Final state: $\neg\mathtt{v1} \wedge \neg\mathtt{v2} \wedge \cdots \wedge \neg\mathtt{v16} \wedge \mathtt{v17} \wedge \neg\mathtt{v18} \wedge \cdots \wedge \neg\mathtt{v33}$
- Transition relation: $\mathtt{R}((\mathtt{v1},\ldots,\mathtt{v33}),(\mathtt{v1}',\ldots,\mathtt{v33}')) = \mathit{complicated\ term}$

# BDD approach to Peg Solitaire (continued)

- Transition relation: $\mathtt{R}((\mathtt{v1},\ldots,\mathtt{v33}),(\mathtt{v1}',\ldots,\mathtt{v33}')) = \mathit{complicated\ term}$
- How is *complicated term* derived?
  - make a list of the moves
  - write a program (in `C`, `C++`, `ML` etc) to create BDD directly
  - using BDD package operations for
    - creating variables
    - negation
    - conjunction
    - disjunction
- Error prone and tedious!

# A generalized version of solitaire

- A board is a finite subset of points in a 2-D lattice

```
 .     .     .
 .     .     .
 .     .     .
(0,2) (1,2) (2,2) . . .
(0,1) (1,1) (2,1) . . .
(0,0) (1,0) (2,0) . . .
```

```
SolitaireBoard =  {            (2,6), (3,6), (4,6),
                               (2,5), (3,5), (4,5),
                  (0,4), (1,4), (2,4), (3,4), (4,4), (5,4), (6,4),
                  (0,3), (1,3), (2,3), (3,3), (4,3), (5,3), (6,3),
                  (0,2), (1,2), (2,2), (3,2), (4,2), (5,2), (6,2),
                               (2,1), (3,1), (4,1),
                               (2,0), (3,0), (4,0)}

SolitaireInit state =
  (state = SolitaireBoard - {(3,3)})

SolitaireFinal state = (state = {(3,3)})
```

- Moves are a subset of `{N n, S n, E n, W n, NE n, SW n, SE n, SW n}` where

```
N  n (x,y) = (x,  y+n)   S  n (x,y) = (x,  y-n)   E  n (x,y) = (x+n,y)     W  n (x,y) = (x-n,y)
NE n (x,y) = (x+n,y+n)   SW n (x,y) = (x-n,y-n)   SE n (x,y) = (x+n,y-n)   NW n (x,y) = (x-n,y+n)

SolitaireMoves = {N 1, S 1, E 1, W 1}
```

- Transition relation

```
Trans board moves (state, state') = ∃p q r ∈ board. (∃move ∈ moves. (q = move p) ∧ (r = move q)) ∧
                                                    (p ∈ state) ∧ (q ∈ state) ∧ ¬(r ∈ state) ∧
                                                    (state' = (state — {p,q}) ∪ r)
```

## A ‘point tool’ for generalized solitaire

- A puzzle is defined by
  - a board
  - a set of moves
  - an initial state
  - a final state
- Tool is to
  - find a solution to any puzzle, if one exists
    - solution is a sequence of transitions from initial to final state (a trace)
  - report if no solution exists and return set of reachable end states
    - `state` is an end state if it has no successors in the transition relation

      `¬∃state'. Trans board moves (state, state')`
    - can then pick a reachable terminal state to make a solvable puzzle

# PuzzleTool as an inference rule

- Derived rules for puzzle designers

$$\frac{\vdash \texttt{Board} = \{\cdots\} \qquad \vdash \texttt{Moves} = \{\cdots\} \qquad \vdash \texttt{Init} = \{\cdots\}}{\vdash \texttt{ReachableEndStates(Board,Moves,Init)} = \{\cdots\}}$$

$$\frac{\vdash \texttt{Board} = \{\cdots\} \qquad \vdash \texttt{Moves} = \{\cdots\} \qquad \vdash \texttt{Init} = \{\cdots\}}{\vdash \texttt{SolutionTrace(Board,Moves,Init)} = [\cdots]}$$

- Looks like logic rules, but have *BDD Inside*
  - actual implementation not packaged this way

# Triv: a trivial example

- Definition of the puzzle Triv

```
TrivBoard = {(0,4),                         OOOO       XXXX
             (0,3),                         XXXX       OOOO
             (0,2),                         OOOO  ==>  OOOO
             (0,1),                         XXXX       OOOO
             (0,0)}                         XXXX       OOOO

TrivMoves = {N 1, S 1}

TrivInit state = (state = {(0,0),(0,1),(0,3)})

TrivFinal state = (state = {(0,4)})
```

- Use standard BDD methods to deduce the solution trace

  `[[F;T;F;T;T]; [F;T;T;F;F]; [T;F;F;F;F]]`

- Rewrite with inverse of boolean representation to get

  `[{(0,3), (0,1), (0,0)}; {(0,3), (0,2)}; {(0,4)}]`

- Same method solves Solitaire

## Another puzzle: Hex Solitaire

- Invented by MJCG!

```
                _____
               /     \
         _____/ XXXXX \_____
        /     \       /     \
  _____/ XXXXX \_____/ XXXXX \_____
 /     \       /     \       /     \
/ XXXXX \_____/ XXXXX \_____/ XXXXX \
\       /     \       /     \       /
 \_____/ XXXXX \_____/ XXXXX \_____/
 /     \       /     \       /     \
/ XXXXX \_____/       \_____/ XXXXX \
\       /     \       /     \       /
 \_____/ XXXXX \_____/ XXXXX \_____/
 /     \       /     \       /     \
/ XXXXX \_____/ XXXXX \_____/ XXXXX \
\       /     \       /     \       /
 \_____/ XXXXX \_____/ XXXXX \_____/
       \       /     \       /
        \_____/ XXXXX \_____/
              \       /
               \_____/
```

```
====>
```

```
                _____
               /     \
         _____/       \_____
        /     \       /     \
  _____/       \_____/       \_____
 /     \       /     \       /     \
/       \_____/       \_____/       \
\       /     \       /     \       /
 \_____/       \_____/       \_____/
 /     \       /     \       /     \
/       \_____/ XXXXX \_____/       \
\       /     \       /     \       /
 \_____/       \_____/       \_____/
 /     \       /     \       /     \
/       \_____/       \_____/       \
\       /     \       /     \       /
 \_____/       \_____/       \_____/
       \       /     \       /
        \_____/       \_____/
              \       /
               \_____/
```

# Hex Solitaire as an instance of generalized solitaire

- Board is a different subset

```
HexSolitaireBoard =
{                               (2,8),
                        (1,7),         (3,7),
                 (0,6),         (2,6),         (4,6),
                        (1,5),         (3,5),
                 (0,4),         (2,4),         (4,4),
                        (1,3),         (3,3),
                 (0,2),         (2,2),         (4,2),
                        (1,1),         (3,1),
                                (2,0)}
```

```
HexSolitaireInit state =
 (state = HexSolitaireBoard - {(2,4)})
```

```
HexSolitaireFinal state = (state = {(2,4)})
```

```
HexSolitaireMoves = {N 2, S 2, NE 1 , SW 1 , SE 1, NW 1}
```

- Points in same row are 2 horizontal units apart
- Points in same column are 1 vertical unit apart
- Joe Hurd has a fiendishly clever representation – ask him for details!

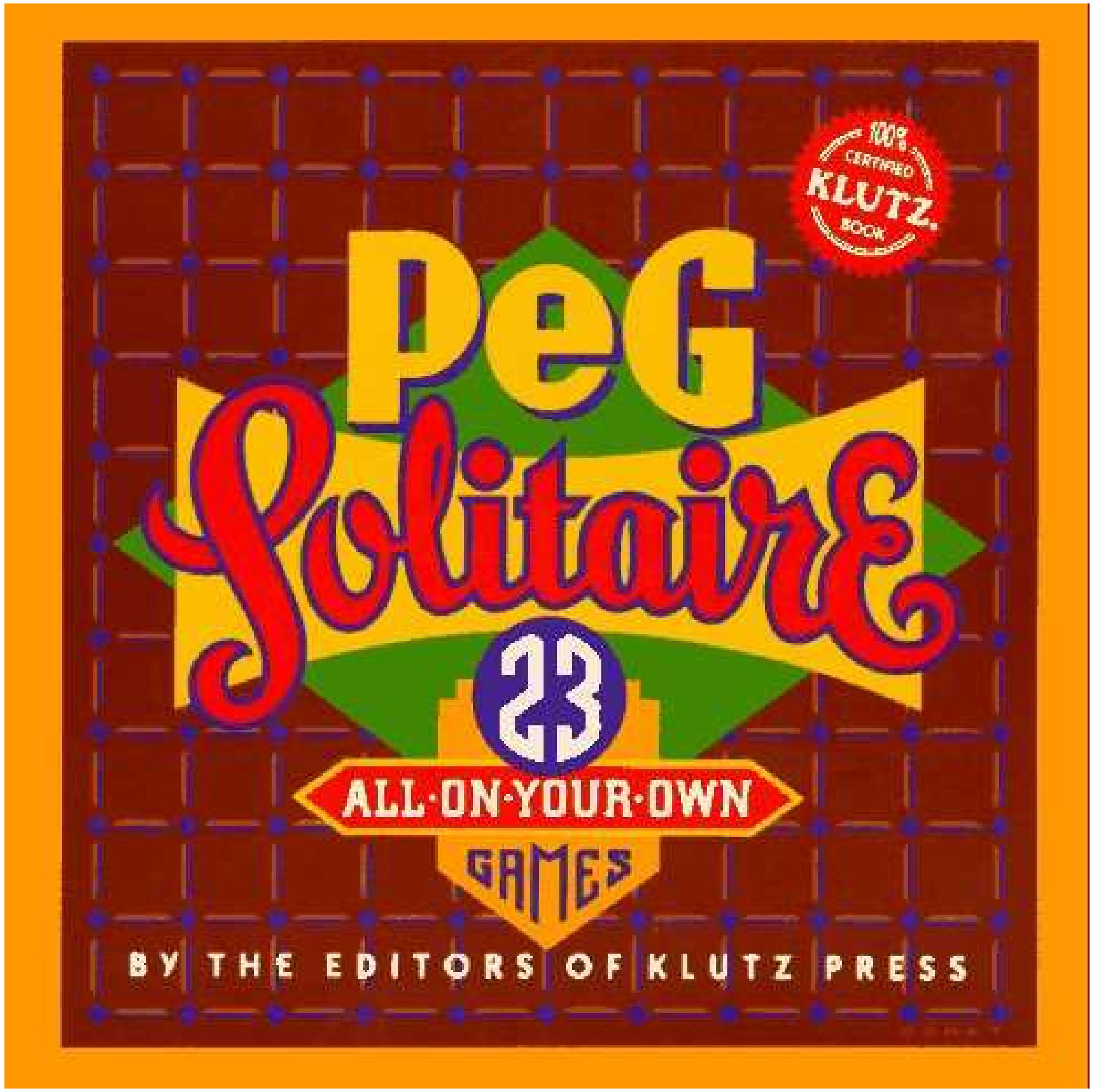
## Finding a solution to HexSolitaire

- `HexSolitaire` has no solution!
  - 660 reachable end states (ignoring symmetry)
- No one-point solutions
  - `{(2,4),(4,4)}` is an example reachable end state

# Concluding thoughts

- Can use proof to build tools
- Need simplifiers and execution engines as well as theorem provers
- Merging computation and deduction
  - advocated by Kowalski and Hayes in early 1970s (logic programming)
  - cycles to spare (2.2GHz etc) $\Longrightarrow$ renaissance of their ideas?

## Business idea – the "killer ap" ?




Volume 2

Volume 3

⋮

$$$