Integrating SAT Solvers with ACL2 (part 1)

Erik Reeber

1/19/05

Overview

- Overview of SAT Solving
- Motivation
- Decidable Subset of ACL2
- Converting ACL2 into CNF
- Conclusion
- General Mechanism For Integrating External Tools (Discussion)

Satisfiability (SAT) Solving

- Does a formula composed of existentially quantified Boolean variables have a satisfying instance?
  - e.g. $\exists x,y,z: x \wedge (y \vee z) \wedge (\neg x \vee \neg z)$
- A SAT solver either:
  - Finds a satisfying instance of the variables
    - e.g. {x<=true, y<=true, z<=false}
  - States that no such instance exists
    - e.g. $\exists x,y,z: x \wedge (y \vee z) \wedge (\neg x \vee \neg z) \wedge z$
  - Fails to finish due to space or time limitations

Conjunctive Normal Form (CNF)

- The standard input format for most SAT solvers is CNF
- In CNF a **formula** is a conjunction of clauses.
- A **clause** is a disjunction of literals
- A **literal** is either a boolean variable or its negation
  - e.g. $\exists x,y,z: x \wedge (y \vee z) \wedge (\neg x \vee \neg z)$

SAT Algorithms

- Davis-Putnam Algorithm (1961)
- DIMACS Annual SAT solving competition
- Chaff
  - Matthew W. Moskewicz, “Chaff: Engineering an Efficient SAT Solver”
- Many applications
  - Hardware Verification: EUCLID, Forte, etc.
  - Planning: Ccalc
  - Graph coloring, cryptography, scheduling, etc.

Motivation

- Contrasting strengths of SAT solving and the ACL2 theorem prover
  - SAT completely automatic & provides counter examples
- SAT solving is easy to formalize in ACL2
  - Existentially quantified formula is the inverse of a universally quantified formula
- Leverage work of those outside ACL2 community
  - Standard input format

SAT & ACL2: A Good Fit?

- Strengths of each are weaknesses of the other
- SAT input format can be formalized easily into ACL2
  - e.g. $\exists x,y,z: x \wedge (y \vee z) \wedge (\neg x \vee \neg z)$ is false iff
  - (not (and x (or y z) (or (not x) (not z))) is an ACL2 theorem.
- Build on the work of other groups
  - SAT solvers continue to improve
  - Input format unlikely to change

Examples

- De Morgan’s Law example
- A simple finite state machine
- Revisit the f74181 ALU
- Verifying a little Verilog Component

De Morgan’s Law

```
(defun unary-and (n x)
  (if (zp n)
      nil
    (and (car x) (unary-and (1- n) (cdr x)))))
(defun unary-or (n x)
  (if (zp n)
      nil
    (or (car x) (unary-or (1- n) (cdr x)))))
(defun not-list (n x)
  (if (zp n)
      nil
    (cons (not (car x)) (not-list (1- n) (cdr x)))))
(thm
 (iff (not (unary-or 2 a))
      (unary-and 2 (not-list 2 a)))
 :hints (("Goal" :sat nil)))
```

De Morgan’s Law Output

```
[Note:  A hint was supplied for our processing of the goal above.
Thanks!]
Eliminating Destructors... numvars: 6
rewrites removed
destructor-elimination complete
Done Elimination Destructors... numvars: 10
Creating zChaff file
Starting printing: 9
Calling zchaff
```

A Counterexample was found:

A: (NIL NIL)

```
; cpu time (non-gc) 0 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  0 msec user, 0 msec system
; real time  370 msec
; space allocation:
;  3,050 cons cells, 46,280 other bytes, 0 static bytes

ACL2 Error in ( THM ...):
```

A 10-digit Decimal Counter

```
(defun n-bleq (n x y)
  (if (zp n)
      t
    (and (iff (car x) (car y))
         (n-bleq (1- n) (cdr x) (cdr y)))))

(defun increment (n x)
  (if (zp n)
      nil
    (if (car x)
        (cons nil (increment (1- n) (cdr x)))
      (cons t (cdr x)))))

(defun next_digit_counter_state (x)
  (if (n-bleq 4 x '(t nil nil t))
      (list '(nil nil nil nil) t)
    (list (increment 4 x) nil)))
```

```
(defun next_counter_state (n x)
  (let* ((curr_d_out (next_digit_counter_state (car x)))
         (curr_d_val (car curr_d_out))
         (curr_d_reset (cadr curr_d_out)))
    (if (zp n)
        nil
      (if curr_d_reset
          (cons curr_d_val (next_counter_state (1- n) (cdr x)))
        (cons curr_d_val (cdr x))))))
```

FSM (continued)

```
(defun valid_digit (a)
  (let ((a1 (cadr a))
        (a2 (caddr a))
        (a3 (cadddr a)))
    (not (and a3 (or a2 a1)))))

(defun valid_digits (n x)
  (if (zp n)
      (not (consp x))
    (and (valid_digit (car x))
         (valid_digits (1- n) (cdr x)))))
```

```
(defthm counter_invariant
  (implies
   (valid_digits 10 x)
   (valid_digits 10 (next_counter_state 10 x)))
  :hints (("Goal" :sat nil)))
;; 0.05s CNF, 0.01s zChaff
```

FSM (continued)

```
;; Run the counter for n cycles
(defun dec_counter (n init-st)
  (if (zp n)
      init-st
    (next_counter_state 10 (dec_counter (1- n) init-st))))

;; Here's a theorem that requires induction...
;; We want valid_digits after n cycles.
(thm
 (implies (valid_digits 10 init-st)
          (valid_digits 10 (dec_counter n init-st))))
```

F74181 ALU

- Performs xor, addition, and some other ops
- ~70 assign statements
- Specification:

```
(defun xor (a b)
  (if a (not b) b))

(defun b-carry (a b c)
  (if a (or b c) (and b c)))
```

```
(defun v-adder (n c a b)
  (if (zp n)
      (list c)
    (cons (xor c (xor (car a) (car b)))
          (v-adder (1- n)
                   (b-carry c (car a) (car b))
                   (cdr a)
                   (cdr b)))))
```

F74181 ALU (continued)

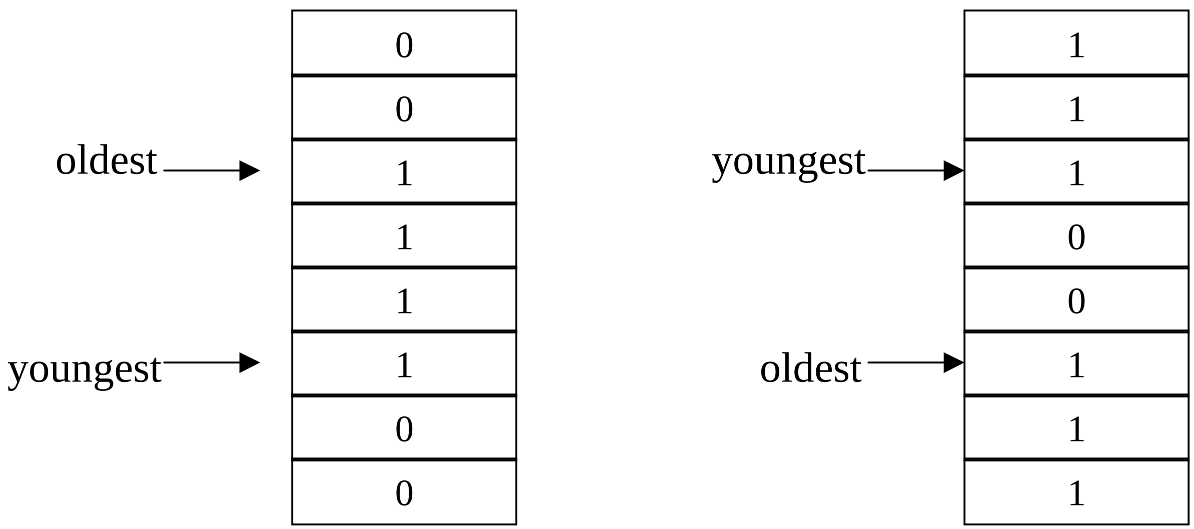
```
;  We now prove that the 74181 can
;; implement an exclusive-or function.
(thm (let* ((s (list nil t t nil))
            (m (list t)))
       (true-bvp
        (bv-eq
         4
         (f74181-f c~ a b m s)
         (bv-xor 4 a b))))
     :hints (("Goal" :sat nil)))
;; 0.06s CNF, 0.01s zChaff (247 Variables)
```

```
;  We state and prove that the 74181 can add.
(thm (let* ((s (list t nil nil t))
            (m (list nil))
            (c~ (list (not cin)))
            (f (f74181-f c~ a b m s))
            (cout~ (f74181-cout~ c~ a b m s)))
       (true-bvp (bv-eq 5 (a-n 4 f (bv-not 1 cout~))
                          (v-adder 4 cin a b))))
     :hints (("Goal" :sat nil)))
;; 0.11s CNF, 0.01s zChaff (311 Variables)
```

A little Verilog Component

```
module dt_lsq_dsn_valid_blocks
  (valid_block_mask, youngest, oldest, empty);

   output [7:0] valid_block_mask;
   input [2:0]  youngest;
   input [2:0]  oldest;
   input        empty;
   wire [7:0]   youngest_set_up, oldest_set_down;
   wire         youngest_lt_oldest;

   assign youngest_set_up =
     youngest[2] &   youngest[1] &  youngest[0] ? 8'b11111111 :
     youngest[2] &   youngest[1] & ~youngest[0] ? 8'b01111111 :
     youngest[2] &  ~youngest[1] &  youngest[0] ? 8'b00111111 :
     youngest[2] &  ~youngest[1] & ~youngest[0] ? 8'b00011111 :
     ~youngest[2] &  youngest[1] &  youngest[0] ? 8'b00001111 :
     ~youngest[2] &  youngest[1] & ~youngest[0] ? 8'b00000111 :
     ~youngest[2] & ~youngest[1] &  youngest[0] ? 8'b00000011 :
                                                8'b00000001;

assign oldest_set_down =
     oldest[2] &   oldest[1] &  oldest[0] ? 8'b10000000 :
     oldest[2] &   oldest[1] & ~oldest[0] ? 8'b11000000 :
     oldest[2] &  ~oldest[1] &  oldest[0] ? 8'b11100000 :
     oldest[2] &  ~oldest[1] & ~oldest[0] ? 8'b11110000 :
     ~oldest[2] &  oldest[1] &  oldest[0] ? 8'b11111000 :
     ~oldest[2] &  oldest[1] & ~oldest[0] ? 8'b11111100 :
     ~oldest[2] & ~oldest[1] &  oldest[0] ? 8'b11111110 :
                                          8'b11111111;

assign youngest_lt_oldest =
     oldest[2] & ~youngest[2] |
     oldest[2] & oldest[1] & ~youngest[1] |
     oldest[2] & oldest[0] & ~youngest[1] & ~youngest[0] |
     oldest[2] & oldest[1] & oldest[0] & ~youngest[0] |
     oldest[1] & ~youngest[2] & ~youngest[1] |
     oldest[1] & oldest[0] & ~youngest[2] & ~youngest[0] |
     oldest[0] & ~youngest[2] & ~youngest[1] & ~youngest[0];

   assign valid_block_mask =
     empty ?              8'd0 :
     youngest_lt_oldest ? youngest_set_up | oldest_set_down :
                          youngest_set_up & oldest_set_down;
endmodule // dt_lsq_dsn
```

A little Verilog Component (cont)



A little Verilog Component (cont)

```
(defun make_valid_mask (n youngest oldest ans)
  (cond ((zp n) ans)
        ((car (bv-eq 3 youngest oldest))
         (bv-or 8 ans (bv-lshift 8 3 (bv-const 8 1) oldest)))
        (t
         (make_valid_mask
          (1- n) youngest (increment 3 oldest)
          (bv-or 8 ans (bv-lshift 8 3 (bv-const 8 1)
                 oldest))))))

(defun valid_blocks (youngest oldest empty)
  (if (car empty)
      (bv-const 8 0)
    (make_valid_mask 8 youngest oldest (bv-const 8 0))))
;; 0.27s to convert to CNF, 0.02s to prove in zChaff.
```

```
(thm (true-bvp
      (bv-eq 8 (valid_blocks youngest oldest empty)
             (car (|acl2-dt_lsq_valid_blocks|
                       youngest oldest empty))))
     :hints (("Goal" :sat nil)))
```

Decidable Fragment

- We have defined a fragment of ACL2 for which our conversion procedure is decidable
- Why?
  - Gives a formal description of the type of models that on which our algorithm performs well.
  - May help future automation
  - Encourages complete automation (see Results)

Type Formals

- The type formals of an ACL2 function is a subset of its formals.
- The type formals of **if**, **consp**, **car**, **cdr**, and **cons** is the empty set.
- For any other primitive or undefined function, the type formals is the complete set of formals
- For a defined function *f*, the type formals of *f* is the minimum subset that satisfies the following restrictions. For any formal *a* of *f*:
  - If *a* appears in the measure of *f*, then *a* is in the set of type formals
  - For every type formal *b* of every function called in the definition of *f*. If *a* is used in an expression to compute *b*, then *a* is in the set of type formals.

Decidable Fragment

- An ACL2 expression E is in our decidable fragment if:
  - For every formal *b* of a function called in E: If *b* is a type formal, then every sub-expression of E which computes **b** evaluates to a constant.
- We compute type formals during function definition
  - In theory this requires quadratic time in the number of formals in a mutually recursive nest
  - In practice, it requires a couple passes.
- Determining whether an expression is decidable, given this info, is linear.

Decidable Subset Example

;; Type Formals and expressions in blue

```
(defun unary-and (n x)
  (if (zp n) t (and (car x) (unary-and (1- n) (cdr x)))))

(defun unary-or (n x)
  (if (zp n) nil (or (car x) (unary-or (1- n) (cdr x)))))

(defun not-list (n x)
  (if (zp n) nil (cons (not (car x))
      (not-list (1- n) (cdr x)))))

(thm (iff (not (unary-or 2 a))
          (unary-and 2 (not-list 2 a)))
 :hints (("Goal" :sat nil)))
```

Results---Performance Comparison

N	Example	ACL2	BDD	SAT
1	4 Adder Assoc	166.72s	0.02s	0.17s
2	32 Adder Assoc	****	0.55s	2.38s
3	200 Adder Assoc	****	6.91s	56.02s
4	32x6 Shift Zeros	106.54s	4.66s	3.27s
5	64x7 Shift Zeros	****	759.79	23.64
6	32x4 Added Shift	****	3.55s	4.13s
7	64x6 Added Shift	****	507.33s	136.13s
8	100 Digit Dec Inv	****	4.53s	11.88s

Results---Lines of Code Comparison

N	Example	Model	ACL2	BDD	SAT
1	4 Adder Assoc	21	17	25	4
2	32 Adder Assoc	21	17	42	4
3	200 Adder Assoc	21	17	202	4
4	32x6 Shift Zeros	34	53	60	6
5	64x7 Shift Zeros	34	53	65	6
6	32x4 Added Shift	44	58	71	4
7	64x6 Added Shift	44	58	77	4
8	100 Digit Dec Inv	36	44	280	4