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. $∃x,y,z: x \land (y \lor z) \land (\neg x \lor \neg z)$

- A SAT solver either:
  - Finds a satisfying instance of the variables
    - e.g. $\{x:=\text{true}, y:=\text{true}, z:=\text{false}\}$
  - States that no such instance exists
    - e.g. $∃x,y,z: x \land (y \lor z) \land (\neg x \lor \neg z) \land z$
  - 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. $\exists x,y,z: x \land (y \lor z) \land (\neg x \lor \neg z)$
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. $\exists x,y,z: x \land (y \lor z) \land (\neg x \lor \neg z)$ is false iff
  - $(\text{not} \, (\text{and} \, x \, (\text{or} \, y \, z) \, (\text{or} \, (\text{not} \, x) \, (\text{not} \, z)))$ 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.]

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))))))
(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)
module dt_lsq_dsn_valid_blocks
(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 =

assign oldest_set_down =

assign youngest_lt_oldest =

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

<table>
<thead>
<tr>
<th>N</th>
<th>Example</th>
<th>ACL2</th>
<th>BDD</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4 Adder Assoc</td>
<td>166.72s</td>
<td>0.02s</td>
<td>0.17s</td>
</tr>
<tr>
<td>2</td>
<td>32 Adder Assoc</td>
<td>****</td>
<td>0.55s</td>
<td>2.38s</td>
</tr>
<tr>
<td>3</td>
<td>200 Adder Assoc</td>
<td>****</td>
<td>6.91s</td>
<td>56.02s</td>
</tr>
<tr>
<td>4</td>
<td>32x6 Shift Zeros</td>
<td>106.54s</td>
<td>4.66s</td>
<td>3.27s</td>
</tr>
<tr>
<td>5</td>
<td>64x7 Shift Zeros</td>
<td>****</td>
<td>759.79</td>
<td>23.64</td>
</tr>
<tr>
<td>6</td>
<td>32x4 Added Shift</td>
<td>****</td>
<td>3.55s</td>
<td>4.13s</td>
</tr>
<tr>
<td>7</td>
<td>64x6 Added Shift</td>
<td>****</td>
<td>507.33s</td>
<td>136.13s</td>
</tr>
<tr>
<td>8</td>
<td>100 Digit Dec Inv</td>
<td>****</td>
<td>4.53s</td>
<td>11.88s</td>
</tr>
</tbody>
</table>
## Results---Lines of Code Comparison

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