Integrating SAT Solvers with ACL2 (part 3)

Erik Reeber
2/2/05
Review of Part 1

• SAT Solvers
  – Find satisfying instances of Boolean variables in conjunctive normal form
  – Used as an alternative to BDDs in fully automated hardware verification tools

• Decidable Fragment of ACL2
  – list structures and unrollable functions
  – detection algorithm
  – Can express hardware invariants
Review of Part 2

• Conversion to BC-CNF
  – Create a clause list that is unsatisfiable only if the original property is valid
  – Create variables using \texttt{bceq} literals
  – Remove functions other than \texttt{car}, \texttt{cdr}, and \texttt{consp} (and one \texttt{bceq} literal)

• $\Gamma$
  – Find relevant components of each variable
  – Requires one pass through the clause list
Substituting BCEQ for BCEQ

- 0: (bceq 4 (f (g x)))

- Create a variable for (g x)
  - 1: (and (bceq 4 (f 5)) (equal 5 (g x)))

- Now substitute bceq for equal
  - 2: (and (bceq 4 (f 5)) (bceq 5 (g x)))

- If an instance satisfies 1, then it satisfies 2.
  - Therefore if no instance satisfies 2 then no instance satisfies 1 (soundness)

- In decidable fragment bceq should be enough
Justifying $\Gamma$

• Too large of a $\Gamma$ leads to inefficiency
  – Create clauses we don’t need

• Too small of a $\Gamma$ leads to spurious counter examples
Motivating Example

• (car 6) is in $\Gamma$ by propagation
  – If not, a spurious counter example would be generated
• If the (car 7) clause were deleted, (car 6) will be in $\Gamma$, but actually be irrelevant
• If the (car 5) clause were deleted, (car 6) will not be in $\Gamma$ and it will be irrelevant
Overview

• Destructor Elimination
• Removing iff
• Results
• Conclusion
• General Mechanism For Integrating External Tools (Discussion)
Example

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

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

;; The (not (not x)) == x
(thm (n-bleq 2 (not-list 2 (not-list 2 x)) x)
    :hints ("Goal" :sat nil))
Destructor Elimination (cont)

- Number all the needed **consp** and non-**nil** expressions
  - These are the new variable numbers

- Add clauses from list structure axioms
  - \((\text{car } x) \rightarrow (\text{consp } x)\)
  - \((\text{cdr } x) \rightarrow (\text{consp } x)\)
  - \((\text{consp } x) \rightarrow x\)
Example

1: ((nil nil nil 1) ((nil nil nil 2) nil nil nil) nil nil)
   => no list structure axioms
2: (nil nil nil 3)
   => none
3: ((nil nil nil 4) ((nil nil nil 5) 6 nil) 7 nil)
   => 4->6, 5->6, and 6->7 (6->empty->7: we continue the chain until we hit a number or run out of things for which to look)
4: ((nil nil nil 8) ((nil nil nil 9) 10 nil) 11 nil)
   => 8->10, 9->10, 10->11
5: (nil nil nil 12)
   => none
6: (nil nil nil 13)
   => none
Destructor Elimination (cont)

• Create new clauses from each old clause
  – For non-bceq literals, sub in the new variable for the component expression
  – Turn bceq clauses into multiple iff clauses, guided by the first argument
Example

- Not based on previous example

\[\Gamma_1 = (\text{nil} \ (\text{nil} \ \text{nil} \ 4 \ 5) \ 6 \ 7) \text{ and} \]
\[\Gamma_2 = ((\text{nil} \ \text{nil} \ \text{nil} \ 8) \ (\text{nil} \ \text{nil} \ 9 \ 10) \ 11 \ 12) \]
\[\Gamma_3 = ((\text{nil} \ \text{nil} \ \text{nil} \ 13) \ \text{nil} \ \text{nil} \ \text{nil} \ 14) \]

\[
(\text{nand} \\
... \\
(\text{or} \ (\text{bceq} \ 1 \ 2) \ 3) \\
=> \\
(\text{nand} \\
... \\
(\text{or} \ (\text{iff} \ 4 \ 9) \ 14) \\
(\text{or} \ (\text{iff} \ 5 \ 10) \ 14) \\
(\text{or} \ (\text{iff} \ 6 \ 11) \ 14) \\
(\text{or} \ (\text{iff} \ 7 \ 12) \ 14))
\]
Example

• In our actual example:
  – The `(bceq (cddr 4) 'nil)` was removed
  – No `bceq` expressions were split

• Result shown on next slide
Example

(nand
  (or (not 4) 7)
  (or (not 5) 6)
  (or (not 6) 7)
  (or (not 8) 11)
  (or (not 9) 10)
  (or (not 10) 11)
  (not 3)
  11
  (iff 8 (not 1))
  10
  (iff 9 (not 2))
  7
  (iff 4 (not 8))
  6
  (iff 5 (not 9))
  (or (iff 12 1) (not 4))
  (or (iff 12 (not 1)) 4)
  (or (iff 12 2) (not 5))
  (or (iff 12 (not 2)) 5)
  (or (iff 3 't) (not 12) (not 13))
  (or (iff 3 'nil) (not 12) 13)
  (or (iff 3 'nil) 12)))
Removing \textit{iff}

- We now remove \textit{iff}
  - (iff \(x\) \(y\)) => (and (or (not \(x\)) \(y\)) (or \(x\) (not \(y\))))
  - (iff \(x\) 't) => x
  - (iff \(x\) 'nil) => (not \(x\))

- Once we remove the \textit{iff} expressions we are in \textit{CNF}.

- Final example shown on next slide
Example

\[(\text{nand} \ (\text{or} \ (\text{not} \ 4) \ 7) \ (\text{or} \ (\text{not} \ 5) \ 6) \ (\text{or} \ (\text{not} \ 6) \ 7) \ (\text{or} \ (\text{not} \ 8) \ 11) \ (\text{or} \ (\text{not} \ 9) \ 10) \ (\text{or} \ (\text{not} \ 10) \ 11) \ (\text{not} \ 3) \ 11) \ (\text{or} \ (\text{not} \ 8) \ (\text{not} \ 1)) \ (\text{or} \ 8 \ 1) \ 10) \ (\text{or} \ (\text{not} \ 9) \ (\text{not} \ 2)) \ (\text{or} \ 9 \ 2) \ 7)
\]
Optimizations

- During BC-CNF conversion: Hash tables used to avoid creating variables for the same expression.
- During Destructor Elimination: Singleton \texttt{bceq} clauses can be deleted and treated as rewrite rules
  - Not entirely implemented
  - Use virtual pointers in $\Gamma$ construction
  - Once showed significant performance improvement, but may no longer be necessary.
Performance

- Opening up functions can lead to an explosion

  \[(\text{thm } (\text{iff } (\text{not } (\text{unary-or } 1000 \ a)))
    (\text{unary-and } 1000 \ (\text{not-list } 1000 \ a)))
  \]

  \[\text{:hints } (("\text{Goal}" :\text{sat} \ \text{nil}))\]

  - Takes 168.61 s to convert to CNF (3003 variables),
  - zChaff took 0.14s to prove.

- The rest of the conversion process is linear

- SAT solving can be exponential in the number of variables, but in practice is not.
## Results---Conversion Decomposition

<table>
<thead>
<tr>
<th>N</th>
<th>Example</th>
<th>BC-CNF</th>
<th>D-Elim</th>
<th>Output</th>
<th>Solving</th>
<th>Vars</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4 bit adder</td>
<td>0.00s</td>
<td>0.00s</td>
<td>0.05s</td>
<td>0.01s</td>
<td>81</td>
<td>0.17s</td>
</tr>
<tr>
<td>2</td>
<td>32 bit adder</td>
<td>0.01s</td>
<td>0.04s</td>
<td>0.06s</td>
<td>2.22s</td>
<td>509</td>
<td>2.38s</td>
</tr>
<tr>
<td>3</td>
<td>200 bit adder</td>
<td>0.53s</td>
<td>1.37s</td>
<td>0.10s</td>
<td>53.75s</td>
<td>4001</td>
<td>56.02s</td>
</tr>
<tr>
<td>4</td>
<td>32x6 Shift Zs</td>
<td>2.54s</td>
<td>0.43s</td>
<td>0.13s</td>
<td>0.01s</td>
<td>7463</td>
<td>3.27s</td>
</tr>
<tr>
<td>5</td>
<td>64x7 Shift Zs</td>
<td>19.08s</td>
<td>2.70s</td>
<td>0.47s</td>
<td>0.05s</td>
<td>29320</td>
<td>23.64s</td>
</tr>
<tr>
<td>6</td>
<td>32x4 Add S</td>
<td>2.45s</td>
<td>0.31s</td>
<td>0.15s</td>
<td>1.0s</td>
<td>6330</td>
<td>4.13s</td>
</tr>
<tr>
<td>7</td>
<td>64x6 Add S</td>
<td>37.31s</td>
<td>6.24s</td>
<td>0.90s</td>
<td>89.54s</td>
<td>54950</td>
<td>136.13s</td>
</tr>
<tr>
<td>8</td>
<td>100 Digit Inv</td>
<td>0.82s</td>
<td>2.72s</td>
<td>1.45s</td>
<td>4.05s</td>
<td>3420</td>
<td>11.88s</td>
</tr>
</tbody>
</table>
## 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>
Conclusion

- SAT solving is a useful technique for verifying hardware
- The strengths of SAT solving contrast those of ACL2
  - Best for small finite theorems
  - Completely automatic
  - Produces counter examples
- We’ve developed a new ACL2 hint which uses an external SAT solver
- This ACL2 hint operates (best) on a well-defined decidable subset of ACL2.
- This subset is large enough to encompass interesting hardware properties (and useful hardware theorems)
External Tool Mechanism

- Desires
  - High performance
  - Powerful
  - No editing ACL2 source code
  - Keep track of dependencies
- Needs of my conversion algorithm
  - Hash-tables
  - Arrays
  - Getprop
  - Ev-fncall
  - I/O & syscall
Proposal

• New hint, :external

• Define a (program-mode) function `tool-fn`
  – `(tool-fn expr state arg₀ arg₁ … argₖ) → (mv erp state val)`

• Use the function in a hint
  – `(defthm rev-n-100
     (bleq (rev-n 100 (rev-n 100 x)) x)
     :hints ((“Goal” :external (sat arg₀ arg₁ … argₖ)))))`

• If `erp` is t, then fail and print error message

• If `erp` is nil, then replace goal `expr` with `val`
  – Tool promises that if `val` is a theorem, then so is `expr`
Proposal (continued)

• Print out external tool dependencies
  – Direct or indirect dependencies?

• Declare dependencies when including books:
  – (include-book “rev-n-100.lisp” :tools (sat))
Thoughts on Proposal

• State modification OK
  – Allows I/O modification to state
  – Untouchables protects internal state
• Can call getprop and trans-eval
• Use ACL2 arrays
• Implement hash-tables