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 usatisfiable only if the original property is valid
  - Create variables using **bceq** literals
  - Remove functions other than **car**, **cdr**, and **consp** (and one bceq literal)
- Γ
  - 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 Γ

- Too large of a Γ leads to inefficiency
  - Create clauses we don’t need
- Too small of a Γ leads to spurious counter examples

Motivating Example

```
…
(bceq (car 7) ‘t)
(bceq 6 7)
(bceq 5 6)
…
(bceq 2 (car 5))
(not 2)
```

- (car 6) is in Γ by propagation
  - If not, a spurious counter example would be generated
- If the (car 7) clause were deleted, (car 6) will be in Γ, but actually be irrelevant
- If the (car 5) clause were deleted, (car 6) will not be in Γ 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
  - (car x) $\rightarrow$ (consp x)
  - (cdr x) $\rightarrow$ (consp x)
  - (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

Γ_1=(nil (nil nil 4 5) 6 7) and
Γ_2=((nil nil nil 8) (nil nil 9 10) 11 12)
Γ_3=((nil nil nil 13) nil nil nil 14)

```
(nand
 …
 (or (bceq 1 2) 3)
=>
(nand
 …
 (or (iff 4 9) 14)
 (or (iff 5 10) 14)
 (or (iff 6 11) 14)
 (or (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 **iff**

- We now remove **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 **iff** expressions we are in CNF.
- Final example 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
 (or (not 8) (not 1))
 (or 8 1)
 10
 (or (not 9) (not 2))
 (or 9 2)
 7
```

```
(or (not 4) (not 8))
(or 4 8)
6
(or (not 5) (not 9))
(or 5 9)
(or (not 12) 1 (not 4))
(or 12 (not 1) (not 4))
(or (not 12) (not 1) 4)
(or 12 1 4)
(or (not 12) 2 (not 5))
(or 12 (not 2) (not 5))
(or (not 12) (not 2) 5)
(or 12 2 5)
(or 3 (not 12) (not 13))
(or (not 3) (not 12) 13)
(or (not 3) 12)
```

Optimizations

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

Performance

- Opening up functions can lead to an explosion

```
(thm  (iff (not (unary-or 1000 a))
           (unary-and 1000 (not-list 1000 a)))
 :hints (("Goal" :sat 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

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

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

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 arg_0 arg_1 … arg_k)$\rightarrow$ (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 arg_0 arg_1 … arg_k)))))
- 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