Checking ACL2 Theorems via SAT Checking

ACL2 Workshop

Grenoble, France

April 9, 2002

Rob Sumners

robert.sumners@amd.com

⌊ **The need for theorem checking...** ⌋

- Basic ACL2 proof strategy: divide-and-conquer
  - In practice, this is really divide-and-divide-and-divide-and-divide-and-divide...
- In order to avoid spending time *proving* non-theorems, we would like to have a tool we could use to automatically check the theorem on some sufficiently-bounded domain of values for the free variables
  - If the theorem fails, we would like an assignment to the free variables witnessing the failure
  - Especially useful in testing inductive invariants
- A theorem checker could also be used in the context of a more general tool to either generate failure witnesses or heuristically prune the paths in search of a proof

⌊ An interface ⌋

- Ideally, before attempting to prove some proposed theorem:

```
(thm <expr>)
```

We would like to first *check* or *test* the theorem:

```
(check-thm (implies <constraint> <expr>))
```

Where `<constraint>` *sufficiently* bounds the free variables in `<expr>`

- Example:

```
(check-thm (implies (and (< (len x) 4)
                         (< (len y) 3))
                    (equal (len (append x y))
                           (+ (len x) (len y)))))
```

- What is "sufficiently bounded"?

⌊ **Our approach...** ⌋

- Basic idea: Translate the constrained theorem into a propositional formula
  - If generated propositional formula is valid, then original ACL2 theorem is valid
    - In practice, the other direction holds as well
  - Use a SAT checker to determine if the propositional formula is valid
  - Allow multiple SAT checkers to be used for engine
  - Translate failure witness for propositional formula into a failure witness for original ACL2 theorem
    - Failure witness (an alist binding the free variables) is double-checked by evaluating the theorem on witness
- The translation consists of two steps: translate the theorem into a simple sublanguage and then *reduce* the theorem to a propositional formula

⌊ Step 1 of the Translation ⌋

- First, translate the history and the proposed ACL2 theorem into a history and theorem in a sublanguage (ST) of ACL2
  - ST histories are built from the primitives `if`, `cons`, `car`, `cdr`, `(quote nil)`
  - ST universe consists of trees where `nil` is the only atom

- The input history and theorem is restricted to be a sublanguage (MDL) of ACL2
  - MDL histories are built from the primitives `if`, `car`, `cdr`, `cons`, `binary-+`, `n-`, `<`, `naturalp`, `symbolp`, `consp`, `equal`, `quote`, ...
  - MDL universe consists of trees where the only atoms are symbols and natural numbers
  - MDL could be extended, but resulting translation could be more expensive
  - Implicit assumption (constraint) of free variables in MDL universe

⌊ Translation of MDL universe ⌋

- Translation from MDL to ST is essentially defined by a translation of the MDL primitives to ST functions

  – This translation is based on mapping of MDL universe to ST universe:

```
(defun mdl-to-tree (x aux)
  (cond
   ((null x) nil)
   ((consp x)
    (st-make-cons (mdl-to-tree (car x) aux)
                  (mdl-to-tree (cdr x) aux)))
   ((naturalp x)
    (st-make-nat (nat-to-list x)))
   (t ;; (symbolp x)
    (st-make-symb
     (nat-to-list (location x (cons t aux)))))))
```

- **aux** parameter is a list of symbols automatically computed from the MDL history

⌊ Translation of MDL primitives ⌋

- For each MDL primitive we define a corresponding ST function

  – e.g. `binary-+` translates to `st-binary-+`:

```
(defun st-coerce-to-nat (x)
  (if (st-naturalp x) x (st-make-nat nil)))

(defun st-binary-+valus (x y)
  (if-cons x (cons nil (st-binary-+valus (cdr x) y)) y))

(defun st-binary-+ (x y)
  (let ((x (st-coerce-to-nat x))
        (y (st-coerce-to-nat y)))
    (st-make-nat (st-binary-+valus (cdr x) (cdr y)))))
```

- We then need to prove that `st-binary-+` is a legal implementation of `binary-+`:

```
(implies (and (good-model-object-p x aux)
              (good-model-object-p y aux))
         (equal (mdl-to-tree (binary-+ x y) aux)
                (st-binary-+ (mdl-to-tree x aux)
                             (mdl-to-tree y aux))))
```

⌊ Step 2 of the Translation ⌋

- We translate the theorem in ST into a propositional formula
  - Propositional formulae (ITEs) are built from variables, booleans, and `(if x y z)` terms
    - Common subterms are constructed uniquely
  - Each free variable in the ST theorem defines a tree of propositional variables – *tree variable positions*(TVPs)
- The translation is an optimized rewriter which:
  - Eliminates `car` and `cdr` applications (may generate *new* TVPs)
  - Reduce the tests of `if` terms to propositional formula
  - Expand all functions (even recursive functions)
    - We must provide mechanisms to avoid unwanted expansion

⌊ Rewriting(evaluation) of ST terms ⌋

```
(defun tfr-eval (term alist ctx fns)
  (if (variablep term)
      (let ((bound (assoc term alist)))
        (if bound (cdr bound) term))
    (case (first term)
      (quote nil)
      (cons (list 'cons (tfr-eval (second term) alist ctx fns)
                        (tfr-eval (third term) alist ctx fns)))
      ((car cdr) (tfr-destruct (first term)
                   (tfr-eval (second term) alist ctx fns)))
      (if (let* ((tst (ite-extract
                         (tfr-eval (second term) alist ctx fns)))
                 (t-ctx (ctx-and ctx tst))
                 (f-ctx (ctx-and ctx (ite-not tst))))
            (cond
             ((ctx-empty f-ctx)
              (tfr-eval (third term) alist t-ctx fns))
             ((ctx-empty t-ctx)
              (tfr-eval (fourth term) alist f-ctx fns))
             (t
              (list 'if tst
                    (tfr-eval (third term) alist t-ctx fns)
                    (tfr-eval (fourth term) alist f-ctx fns))))))
            (otherwise
             (mv-let (formals body)
                 (if (flambdap operator)
                     (mv (lambda-formals operator)
                         (lambda-body operator))
                   (lookup-function operator fns))
               (tfr-eval body
                         (tfr-eval-bind formals (rest term)
                                        alist ctx fns)
                         ctx fns)))))))
```

⌊ Elaborations and Optimizations ⌋

- We need to maintain a context in order to lazily evaluate `if`
  - `ctx-and` is used to extend *ctx* and `ctx-empty` determines if a *ctx* is consistent
  - In our case, a context is a partial assignment of the TVPs which must hold in the current context
    - efficient and (hopefully) sufficient

- Several optimizations in the term representation and evaluation
  - e.g. ITEs and TVPs are constructed uniquely, hash tables for lookup, etc.

- Translation maintains statistics on function expansion to assist in determining where constraints are insufficient
  - The translator also provides depth bounds for each function's "stack"

⌊ Translating ITE to SAT checker ⌋

- In order to reduce the formula given to the SAT checker, we perform an initial simplification which:
  - Iteratively constructs a partial assignment which must hold for any satisfying assignment
  - Reduce the formula under this partial assignment
  - Save the partial assignment to include with any results from SAT checker
    - The `<constraint>` will often reduce to `T`
- We also need to communicate relationship between TVPs (i.e. `(implies (car x) x)`)
- Translation to external SAT checkers involves creation of input file, `sys-call` to run the SAT checker, and parsing of the output file

⌊ Translating SAT results to ACL2 ⌋

- If the SAT check produces a failure witness, the witness will define a (partial) assignment to the propositional TVPs

  – We first translate TVP assignment to a binding of the free variables in the theorem to ST objects

  – We then translate this assignment to a binding of free variables with MDL objects using the inverse mapping `tree-to-mdl`

  – Finally, we double-check the failure witness on the original theorem by evaluating the theorem

- In the case of our internal SAT checker, a partial assignment can be returned which may be useful in analyzing automatically generated theorems

⌊ Example: mutual exclusion ⌋

```
(defun step-state (s f)
  (case s
        (try       (if f 'try 'go))
        (go        'wait)
        (otherwise 'try)))

(defun step-flag (s f)
  (case s
        (try       t)
        (go        nil)
        (otherwise f)))

(defun next (l n)
  (let ((f (car l))
        (s (get-nth n (cdr l))))
    (cons (step-flag s f)
          (set-nth n (step-state s f) (cdr l)))))

(defun no-one-go (l)
  (if (endp l) t
    (and (not (equal (car l) 'go))
         (no-one-go (cdr l)))))

(defun only-one-go (l)
  (and (consp l)
       (if (equal (car l) 'go)
           (no-one-go (cdr l))
         (only-one-go (cdr l)))))

(defun good (l)
  (if (car l)
      (only-one-go (cdr l))
    (no-one-go (cdr l))))
```

⌊ Example continued ⌋

```
(defun boundedp (l k)
  (if (0p k) (not l)
    (and (consp l)
         (member (car l) '(try go wait))
         (boundedp (cdr l) (n- k 1)))))

(defun constrain (l n k)
  (and (consp l)
       (member (car l) '(t nil))
       (boundedp (cdr l) k)
       (naturalp n)
       (< n k)))

(check-thm
  (implies (constrain l n 4)
           (implies (good l)
                    (good (next l n)))))
```

- What makes a good constraint?
  - The constraint should be sufficient for evaluation to terminate (checker provides feedback)
  - The weaker the constraint, the stronger the result
  - A stronger constraint may afford more efficient SAT checking and make failure witnesses easier to comprehend

⌊ **Future Work – guiding SAT** ⌋

- ITE is *natural form* of translation
  - Can asymmetry between test and branches provide hints to decision structure during SAT check?
  - Initial attempts at defining a SAT checker for ITE forms failed because I did not see the relevance of splitting on intermediate nodes
    - natural byproduct of translation to CNF

- The following case split is (roughly) sufficient:
  - `(car l)`, and in the `only-one-go` case, a further split on the location of `'go`, and a case split on `n`

- Work continues on heuristics and user annotation to better direct decisions made in SAT checker

⌊ Future Work – Proof of correctness ⌋

• In some cases it would be useful to actually **prove** theorems using the theorem checker

• The sanctioned approach is to define a meta-function which maps terms to (provably) equivalent terms, but evaluator is limited

```
(defthm theorem-checker-is-correct
  (let* ((fns (assemble-MDL-functions term state))
         (aux (quoted-symbols-in-fns fns)))
    (implies (and (good-mdl-object-alist-p alist aux)
                  (equal (check-thm term) :qed))
             (mdl-eval term alist fns))))
```

• In order to prove this, we will need to prove each step of the translation is correct:

– Translation from MDL functions to ST functions is consistent via `mdl-to-tree`

– Interpretation of term returned by `tfr-eval` is consistent with evaluation of ST functions