Correctness Proof of a BDD Manager in the Context of Satisfiability Checking

Rob Sumners

Advanced Micro Devices
UT/Austin

ACL2 Workshop
October 31, 2000

⌊ **Overview** ⌋

- Initial Concepts/Definitions
  - A short review of Single-Threaded Objects (stobjs)
  - Propositional Satisfiability Checking
    - When is a sat. checker correct? Why is this our goal??
  - Binary Decision Diagrams
- Definitions and Theorem Proving
  - Definition and Use of Simple BDD functions
  - Definition and Proof of Stobj BDD functions
    - Invariant of the BDD manager stobj
- Optimizations, Extensions, and Experiments
- WARNING - There were 431 occurrences of the three letters "bdd" in the paper – my sincerest apologies

⌊ Previous Work ⌋

- [Bryant86] introduced the use of Reduced Ordered Binary Decision Diagrams as a canonical representation of boolean functions

- Numerous extensions/applications

  – "Symbolic X" where $X \in \{$ model checking, equiv. checking, trajectory evaluation, ... $\}$

  – Dynamic variable reordering, Multi-valued DDs, Zero-suppressed DDs, ...

- [Moore94] implemented BDD algorithms in ACL2, Kaufmann then added term-level BDDs to the ACL2 prover

  – triggered by the `:bdd` theorem hint

- [Harrison95] interfaced BDDs to HOL as a derived rule

- [Verma,Goubault-Larrecq00] implemented and verified a BDD implementation in the theorem prover Coq

  – Our approach is similar, but the use of stobjs improves performance significantly

⌊ Single-Threaded Objects (stobjs) ⌋

- User provides declarations that certain objects are single-threaded
  - Single-threadedness is then enforced through syntactic restrictions
    - Restrictions ensure that destructive operations coincide with applicative semantics
    - The ACL2 `state` is a built-in stobj
- Stobj array fields are lists in the logic, but common lisp arrays under-the-hood
  - important for fast access and update
- Stobjs were initially used by Greve, Hardin, and Wilding to develop an efficient hardware simulator in ACL2

⌊ Propositional Terms ⌋

- A propositional *term* is either:
  - A propositional constant – either **`T`** or **`nil`**
  - A propositional variable – represented by a positive integer
  - A decision node – **`(dn test then else)`**
    - where **`test,then,else`** are propositional terms

```
(defun prop-ev (f a)
  (cond ((prop-varp f) (prop-look f a))
        ((atom f) (if f T nil))
        (t (prop-if (prop-ev (test f) a)
                    (prop-ev (then f) a)
                    (prop-ev (else f) a)))))

(defun prop-varp (x) (and (integerp x) (> x 0)))

(defun prop-look (v a)
  (cond ((endp a) nil)
        ((equal v (caar a))
         (if (cdar a) T nil))
        (t (prop-look v (cdr a)))))

(defun prop-if (f g h)  (if f g h))
```

⌊ Satisfiability Checking ⌋

- A propositional satisfiability checker `sat-check` is a function which takes a term and returns `nil` iff for all `a`, `(prop-ev f a) = nil`

  - In ACL2, we verify `sat-check` by defining a function `sat-witness` and prove the following:

```
(defthm sat-check-is-correct
  (if (sat-check f)
      (prop-ev f (sat-witness f))
    (not (prop-ev f a))))
```

- Our goal is to define and verify a sat. checker using our BDD implementation

  - Why?? a sat. checker has a clear and complete statement of correctness, the BDD functions (in my opinion) do not

⌊ Binary Decision Diagrams ⌋

- (Reduced Ordered) *BDD*s are propositional terms which are restricted to satisfy the predicate **`robdd`** below

```
(defun robdd (f)
  (or (booleanp f)                          ;; leaves are T or nil
      (and (consp f)
           (bdd-test> f (then f))           ;; ORDERED
           (bdd-test> f (else f))
           (not (bdd= (then f) (else f))) ;; REDUCED
           (pnatp (test f))                 ;; test is a variable
           (robdd (then f))
           (robdd (else f)))))

(defun bdd= (f g)
  (cond ((and (atom f) (atom g)) (iff f g))
        ((or (atom f) (atom g)) nil)
        (t (and (equal (test f) (test g))
                (bdd=  (then f) (then g))
                (bdd=  (else f) (else g))))))

(defun bdd-test> (f g)
  (or (atom g) (> (test f) (test g))))
```

- Now prove that (RO)BDDs are canonical

```
(defcong bdd= equal (prop-ev f a) 1)

(defthm robdd-not-bdd=-implies-not-prop-ev-=
  (implies (and (robdd f) (robdd g)
                (not (bdd= f g)))
           (not (equal (prop-ev f (robdd-witness f g))
                       (prop-ev g (robdd-witness f g))))))
```

⌊ Proof Strategy ⌋

- Stobj functions are forced to explicitly denote (and return) any updates to the stobj variable
  - Reasoning about stobj functions is analogous to reasoning about state machines
  - The stobj holds the state and functions only return correct values with "well-formed" states and inputs
  - "well-formed" should be an invariant preserved by every stobj update
- Approach:
  - Define Simple stobj-free function counterparts
  - Prove the necessary properties about the Simple functions
  - Prove the Stobj functions are consistent with the Simple functions in well-formed states
    - Prove that well-formed is an invariant of the Stobj functions

⌊ Simple BDD functions ⌋

- Definition and selected properties of the simple `spec` functions

```
(defun eql-spec (f g) (bdd= f g))

(defun var-spec (n) (dn n T nil))

(defun ite-spec (f g h)
  (if (atom f) (if f g h)
    (let ((v (top-var f g h)))
      (let ((then (ite-spec (v-then f v)
                            (v-then g v)
                            (v-then h v)))
            (else (ite-spec (v-else f v)
                            (v-else g v)
                            (v-else h v))))
        (if (bdd= then else) then
          (dn v then else))))))

(defthm ite-spec-returns-robdds
  (implies (and (robdd f) (robdd g) (robdd h))
           (robdd (ite-spec f g h))))

(defthm ite-spec=prop-if-under-prop-ev
  (implies (and (robdd f) (robdd g) (robdd h))
           (equal (prop-ev (ite-spec f g h) a)
                  (prop-if (prop-ev f a)
                           (prop-ev g a)
                           (prop-ev h a)))))
```

⌊ **Reductions of `ite-spec`** ⌋

- Proofs of various reductions for `ite-spec`

  - Allows optimization in the stobj function `ite-bdd`

```
(defthm ite-spec-reduction-1
  (implies (robdd f)
           (bdd= (ite-spec f T nil) f)))

(defthm ite-spec-reduction-2
  (implies (and (robdd g) (robdd h) (bdd= g h))
           (bdd= (ite-spec f g h) g)))

(defthm ite-spec-reduction-3
  (implies (and (robdd f) (robdd g) (robdd h) (bdd= f g))
           (bdd= (ite-spec f g h)
                 (ite-spec f T h))))

(defthm ite-spec-reduction-4
  (implies (and (robdd f) (robdd h) (bdd= f h))
           (bdd= (ite-spec f g h)
                 (ite-spec f g nil))))
```

- Example reduction:

```
(and f f) => (ite f f nil) => (ite f T nil) => f
```

⌊ Stobj BDD functions ⌋

- We now define the stobj-based BDD functions

```
(defun eql-bdd (x y)
  (if (atom x) (and (atom y) (iff x y))
    (and (consp y) (eql (tag x) (tag y)))))

(defun var-bdd (n bdd-mgr) (get-unique n T nil bdd-mgr))

(defun ite-bdd (f g h bdd-mgr)
  (cond ((atom f) (if f (mv g bdd-mgr) (mv h bdd-mgr)))
        ((and (eq g T) (not h))   (mv f bdd-mgr))  ;; redux-1
        ((eql-bdd g h)            (mv g bdd-mgr))  ;; redux-2
        ((eql-bdd f g) (ite-bdd f T   h bdd-mgr))  ;; redux-3
        ((eql-bdd f h) (ite-bdd f g nil bdd-mgr))  ;; redux-4
        (t (let ((entry (find-result f g h bdd-mgr)))
             (if entry (mv (ite-rslt entry) bdd-mgr)
               (seq ((v (top-var f g h))
                     ((then bdd-mgr) (ite-bdd (v-then f v)
                                              (v-then g v)
                                              (v-then h v)
                                              bdd-mgr))
                     ((else bdd-mgr) (ite-bdd (v-else f v)
                                              (v-else g v)
                                              (v-else h v)
                                              bdd-mgr))
                     ((rslt bdd-mgr)
                      (if (eql-bdd then else) (mv then bdd-mgr)
                        (get-unique v then else bdd-mgr)))
                     (bdd-mgr (set-result f g h rslt bdd-mgr)))
                    (mv rslt bdd-mgr)))))))

(defun free-bdd (keep bdd-mgr)
  (let ((bdd-mgr (init-bdd bdd-mgr)))
    (rebuild-bdds keep bdd-mgr)))
```

⌊ Lemmas about Stobj functions ⌋

- Main properties needed about the stobj BDD functions

```
(defthm eql-bdd-is-correct
  (implies (and (uniq-tbl-inv bmr)
                (in-uniq-tbl f bmr)
                (in-uniq-tbl g bmr))
           (iff (eql-bdd f g) (bdd= f g))))

(defthm ite-bdd-preserves-in-uniq-tbl
  (implies (in-uniq-tbl b bmr)
           (in-uniq-tbl b (mv-nth 1 (ite-bdd f g h bmr)))))

(defthm ite-bdd-is-correct
  (implies (and (bdd-mgr-inv bmr)
                (in-uniq-tbl f bmr)
                (in-uniq-tbl g bmr)
                (in-uniq-tbl h bmr)
                (robdd f) (robdd g) (robdd h))
           (mv-let (r nbm) (ite-bdd f g h bmr)
             (and (in-uniq-tbl r nbm)           ;; Step 1
                  (bdd-mgr-inv nbm)             ;; Step 1,2
                  (bdd= r (ite-spec f g h)))))) ;; Step 2
```

- The predicate `uniq-tbl-inv` is implied by `bdd-mgr-inv`, but the weaker assumption in `eql-bdd-is-correct` is necessary for the proof of `ite-bdd-is-correct`

The BDD-manager invariant

```
(defun uniq-tbl-inv (bmr)
  (let ((uniq-lst (flatten (uniq-tbl bmr)))
        (rslt-lst (rslt-tbl bmr)))
    (and (integerp (next-tag bmr))
         (consesp uniq-lst)
         (codes-match (uniq-tbl bmr) 0)
         (no-dup-tags uniq-lst)
         (no-dup-nodes uniq-lst)
         (contained uniq-lst uniq-lst)
         (tags-bounded uniq-lst (next-tag bmr))
         (rslts-contained rslt-lst uniq-lst))))

(defun bdd-mgr-inv (bmr)
  (and (uniq-tbl-inv bmr)
       (ite-results (rslt-tbl bmr))))
```

1. `(codes-match (uniq-tbl bmr) 0)` – Ensures that every BDD node in the chain at address I in the `uniq-tbl` hashes to I. This allows us to reduce the search for a matching node in the `uniq-tbl` to a matching node in the chain at the proper hash-code.
2. `(no-dup-tags uniq-lst)` – No two nodes in the `uniq-tbl` have the same `tag` value. This ensures the uniqueness of tags in the `bdd-mgr`.
3. `(no-dup-nodes uniq-lst)` – No two nodes in the `uniq-tbl` are `bdd=`. This ensures the uniqueness of nodes (w.r.t. `bdd=`) in the `bdd-mgr`.
4. `(contained uniq-lst uniq-lst)` – Ensures that every bdd node in the `uniq-tbl` satisfies the predicate `in-uniq-tbl`. The predicate `(in-uniq-tbl f bmr)` returns `T` iff `f` is embedded in the `uniq-tbl`
5. `(tags-bounded uniq-lst (next-tag bmr))` – Every `tag` of every bdd node is bounded by `next-tag`. This allows the use of `next-tag` as the `tag` value for the next bdd node added without invalidating `no-dup-tags` above.

⌊ Wrapping Up ⌋

```
(defun term->bdd (term bdd-mgr)
  (cond ((prop-varp term)
         (var-bdd term bdd-mgr))
        ((atom term)
         (mv (if term T nil) bdd-mgr))
        (t (seq (((f-bdd bdd-mgr)
                  (term->bdd (test term) bdd-mgr))
                 ((g-bdd bdd-mgr)
                  (term->bdd (then term) bdd-mgr))
                 ((h-bdd bdd-mgr)
                  (term->bdd (else term) bdd-mgr)))
                (ite-bdd f-bdd g-bdd h-bdd bdd-mgr)))))

(defthm term->bdd-is-correct  ;;;; key property
  (implies (bdd-mgr-inv bmr)
           (mv-let (b nbm) (term->bdd f bmr)
              (and (robdd b)
                   (equal (prop-ev b a)
                          (prop-ev f a))))))

(defun bdd-sat? (term bdd-mgr)
  (seq ((bdd-mgr (clear-bdd bdd-mgr))
        ((f-bdd bdd-mgr) (term->bdd term bdd-mgr)))
       (mv (not (eql-bdd f-bdd nil)) bdd-mgr)))

(defthm bdd-sat?-is-sat-checker
  (implies (bdd-mgrp bmr)
           (if (mv-nth 0 (bdd-sat? f bmr))
               (prop-ev f (mv-nth 0 (sat-witness f bmr)))
             (not (prop-ev f a)))))
```

⌊ Optimizations ⌋

- Common Lisp Optimizations
  - Macros instead of (non-recursive) Functions
  - Type declarations (especially fixnum declarations)
  - Efficient function replacements, **`equal => eq`**, **`mod => logand`**, **`* => ash`**, etc.
- Memory Management
  - Conses are expensive – time and space
  - Use a (large) stobj array for allocating nodes
    - drawback: limited array sizes in Common Lisp
- Primitive Complement
  - Support very fast complementation by pointer manipulation
  - Increases normalization of terms and improves usage of result caches

⌊ **Extensions** ⌋

- Dynamic Variable Reordering
  - BDD size is very sensitive to the ordering of the variables
  - May be difficult to determine good ordering statically
  - Many BDD managers implement heuristics for performing sequences of adjacent variable swaps
- Additional Operations
  - Partitioned Image Computation
    - Useful for speeding up image computations needed for model checking
  - Projection
    - Existential quantification of a set of prop. var.s
- Term-Level BDDs
  - Extend BDD proof to terms using encapsulated term evaluator instead of `prop-ev`

⌊ Experiments ⌋

- Implemented an optimized BDD manager in order to permit meaningful comparison with C-compiled BDDs

  – Compared with the CUDD package from Colorado/Boulder compiled with GCC and a hand-translation of the BDD manager also compiled with GCC

- Comparison performed on Urquhart's U-problem (below), multiplication of size N bitvectors, and a random construction

$$x_1 \Leftrightarrow (x_2 \Leftrightarrow ...(x_N \Leftrightarrow (x_1 \Leftrightarrow (x_2 \Leftrightarrow ...(x_{N-1} \Leftrightarrow x_N)...)))...)$$

  – Tests performed on a Sun UltraSparc using GCC -O3 and Franz Allegro Common Lisp; execution times are in seconds:

Problem	Parameter(N)	ACL2	GCC	CUDD
Urquhart	1000	4.3	1.5	2.0
	1200	6.5	2.4	3.0
	1400	9.5	3.8	4.2
multiply	10	1.4	0.3	0.6
	11	4.6	1.2	1.0
	12	15.8	4.5	2.9
random	700	10.1	3.4	4.6
	1000	14.4	4.8	6.5
	1300	13.6	4.4	5.8

⌊ Future Work/Wish List ⌋

- Verify optimized BDD manager functions
- Verify term-level BDD implementation
  - possible use in ACL2 model/invariant checker
- Wish List
  - Attempt all instances of free variables in applications of forward-chaining rules
  - Turn stobj access/update functions into macros
    - This accounted for almost 1/2 of the performance gap between ACL2 and GCC in some cases
    - Turn stobj field storage into simple-vector