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

Rob Sumners

Computer Engineering Research Center
The University of Texas at Austin
robert.sumners@amd.com

Abstract

We present a compositional proof of correctness for a binary decision diagram (BDD) manager used in the context of a propositional satisfiability checker implemented using Single-Threaded Objects (stobjs) in ACL2. The use of stobjs affords the definition of an efficient BDD manager which ensures unique construction, allows constant-time comparison, and caches previously computed results. The use of ACL2 means we can prove that the BDD manager implements the prescribed task of building a normal-form representation of a boolean formula. We divide the proof requirements into (1) showing that a simpler set of BDD functions is correct, and (2) showing that the stobj-based BDD functions return values consistent with these simpler functions. We conclude the paper with a discussion of future extensions and refinements to the BDD manager presented.

1 Introduction

Reduced ordered binary decision diagrams (BDDs) were originally developed as a canonical form for representing boolean functions[3]. Their range of application has grown beyond this largely due to the elegant procedures which exist for performing various operations directly on their structure while maintaining their canonical form. BDDs have been used in equivalence checkers, symbolic model checkers, optimizing compilers, mechanical theorem provers, and other programs which need to manipulate boolean logic terms. In each of these contexts, a program uses BDD nodes allocated and maintained by a BDD manager and the correctness of the program invariably depends on the correctness of the BDD manager.

We present a proof for a straightforward implementation of a BDD manager[1] in ACL2 using Single Threaded Objects (stobjs). Recently, an implementation of BDDs was defined and verified in the theorem prover Coq [13]. The work presented here is similar, but with the use of stobjs and some additional optimizations in ACL2, our implementation is significantly more efficient. Some features of existing BDD managers (most notably, dynamic variable reordering) are absent in our implementation. Yet, we feel the structure of the work presented lends itself to extension and provides a significant step in the direction of formally verifying an optimized, full-featured BDD manager in ACL2. We also note that we prove nothing about the optimality or performance of BDDs in this paper. Indeed, it is well-documented[4] that even in the best-case, BDDs require an exponential (in the number of boolean variables) number of nodes to represent certain functions (e.g. multiplication).

Section 2 presents the verification of a simplified version of the BDD operations. Section 3 then defines the stobj-based BDD functions and shows that these functions return values consistent with their simplified counterparts. We conclude the paper in Section 4 with an outline of possible extensions to the work presented here. We also present in Section 4 some experiments comparing the execution times of an optimized version of the BDD manager we analyze in this paper, a translation of this optimized BDD manager to C, and the popular CU Decision Diagram (CUDD) package[12]. We have omitted, for the sake of presentation, all :hints, :rule-classes, :guard, :stobjs, and :measure declarations. The interested reader should consult the books accompanying this paper (available from the author if necessary) for these details.

1.1 Single-Threaded Objects

In ACL2 versions 2.4 and later, the user can declare certain objects to be single-threaded[10]. This declaration allows the use of destructive operations while maintaining the applicative semantics of ACL2. When an object is declared to be single-threaded, ACL2 places certain syntactic constraints on the use of the object which ensure that only one copy of the object (and only one pointer or reference to the object) exists. With this restriction, destructive updates to the stobj are consistent with the applicative semantics of ACL2 since the old value of the stobj cannot affect subsequent evaluation. A stobj is declared with a `defstobj` form which defines the name of the stobj, the fields of the stobj, the types of the fields, and their initial values. Of particular significance are fields which are declared as array types. ACL2 has some built-in support for applicative arrays using `aref1, aref2, aset1, aset2` but these operations have to maintain an association list with every update since there is no guarantee of single-threadedness. Array fields in stobjs are essentially Common Lisp arrays and thus, in contexts where the single-threaded restrictions are not a significant burden, stobj arrays are preferable to applicative arrays. In the logic, operations on stobjs are translated to corresponding operations on lists defined with `nth` and `update-nth`. We will use stobjs to maintain hash-tables for nodes in BDDs. This allows us to create BDD nodes uniquely and annotate the nodes with tags which afford a fast constant-time check of BDD equivalence. Stobjs are also used to cache the results of previously-computed function evaluations to avoid their recomputation.

2 Simplified BDD Operations

BDDs are a data structure used to represent propositional terms. An implementation of BDDs is sound if the operations performed on BDDs are consistent with the semantics of the corresponding operations on propositional terms. Translating this notion of correctness to an efficient implementation of BDDs is complicated by the likelihood that the efficient implementation only behaves correctly under certain "well-formed" conditions on the inputs. While it is certainly possible to define such conditions, and prove that these conditions are preserved in the implementation, it is difficult to ensure that these conditions are complete. In light of this, we define a simple function which uses the BDD manager for a particular application (in our case, propositional satisfiability checking) which has an obvious and complete statement of correctness we can then verify. It is possible that some future application of the BDD manager may require additional theorems to be proven. Nonetheless, we believe the theorems proven in this work provide a clear basis for analyzing future applications.

We begin by defining the syntax and semantics of propositional terms. Propositional terms (or simply terms) are built from the propositional constants `T` and `nil`, propositional variables $x_1, x_2, ...$ (represented by their natural number subscript), and "decision nodes", built with the ternary[1] constructor function `dn`, of the form `(dn` f g h`)` where f, g, h are terms. The functions `test`, `then`, and `else` are the corresponding access functions of a decision node `(dn test then else)`. The semantics of a propositional term is a function mapping boolean valuations of the propositional

[1]Technically, `dn` takes four parameters where the fourth parameter is a special `tag` value used for efficient comparison in the stobj-based implementation. The semantics of decision nodes are independent of this `tag` field and for the moment we ignore this extra parameter to `dn`

```
(defun pnatp (x)  ;; recognizer for propositional variables
  (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)  ;; we need an ``if'' we can control
  (if f g h))

(defun prop-ev (f a)
  (cond ((pnatp 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)))))

;;;; define function ((sat-check f) -> bool) and
;;;; ((sat-witness f) -> a) such that the following can be shown:

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

Figure 1: Statement of Correctness for sat-checking

variables to either `T` or `nil`. This mapping is defined by the function `prop-ev` in Figure 1. A term f is *satisfiable* iff there exists a valuation a of the propositional variables such that `(prop-ev` f a`)` = `T`. A *satisfiability checker* is a predicate which takes a term and returns `T` iff the term is satisfiable. The correctness of a satisfiability checker `sat-check` is specified in ACL2 by the existence of a function `sat-witness` such that the theorem `sat-check-is-correct` in Figure 1 can be proven. Our goal is to use BDDs to define a simple satisfiability checker and then prove the proper variant of `sat-check-is-correct`.

Reduced ordered binary decision diagrams (BDDs) are propositional terms which satisfy certain restrictive criteria. These criteria are enumerated below and defined in ACL2 by the predicate `robdd` in Figure 2.

1. Every BDD is either a propositional constant or a decision node.
2. For every `(dn` f g h`)`, f is a natural number (i.e. a propositional variable) and g and h are BDDs.
3. For every `(dn` x `(dn` y f g`)` h`)` and `(dn` x f `(dn` y g h`))`, the propositional variables are ordered, e.g. $x > y$.
4. For every `(dn` f g h`)`, it is the case that $g \neq h$.

As we mentioned before, BDDs are *canonical* representations of propositional terms. Specifically, two BDDs are structurally equivalent iff they are semantically equivalent. The relation `bdd=`

```
(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))))))

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

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

(defun robdd (f)
  (or (booleanp f)
      (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 propositional variable
           (robdd (then f))
           (robdd (else f)))))

(defthm robdd-bdd=-saturates-prop-ev-=
  ;; this theorem (combined with the above congruence) ensures
  ;; that BDDs are a canonical form for propositional terms
  (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))))))
```

Figure 2: Definition and properties of BDDs

given in Figure 2 defines structural equivalence for BDDs. Two BDDs f and g are semantically equivalent iff for all valuations a, `(prop-ev` f a`)` = `(prop-ev` g a`)`. The proof that structural implies semantic is straightforward and leads to the congruence in Figure 2. The other direction is shown by defining a witness function `robdd-witness` which takes two structurally inequivalent BDDs and builds a valuation on which they differ under `prop-ev`. Our `robdd-witness` constructs this valuation by consing up the values of the propositional variables encountered along the path of inequivalent nodes in both BDDs eventually reaching `T` and `nil`. After proving a few simple theorems relating variables and BDDs which are independent, ACL2 was able to find a proof to the theorem `robdd-bdd=-saturates-prop-ev-=` which completes the canonical-form argument.

We now consider the simple versions of the BDD operations; these simple versions are defined in Figure 3. The function `var-spec` takes a propositional var `n` and returns a node which splits on `n`. The predicate `eql-spec` takes two BDDs and returns `T` iff they are structurally (and semantically) equivalent. The function `ite-spec` takes three BDDs and constructs a BDD for the generic if-then-else operation. We use "generic" here to mean that any of the sixteen binary boolean operations can be translated to an appropriate call of `ite-spec`. For instance, `(bdd-and f g)` = `(ite-spec f g nil)`, `(bdd-or f g)` = `(ite-spec f T g)`, and `(bdd-not f)` = `(ite-spec f nil T)`. The evaluation of `(ite-spec f g h)` first tests if `f` is an atom and if so returns either `g` or `h` depending on whether `f` is non-`nil`. Otherwise, the `test` variables (the `test` of atoms is defined to be 0) of `f`, `g`, and `h` are compared and the maximum is returned by `top-var`. `ite-spec` is then called recursively

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

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

(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-is-correct
  (implies (and (robdd f) (robdd g) (robdd h))
           (and (robdd (ite-spec f g h))
                (equal (prop-ev (ite-spec f g h) a)
                       (prop-if (prop-ev f a)
                                (prop-ev g a)
                                (prop-ev h a))))))
```

Figure 3: Definition of `spec` operations and relevant properties

on the then and else branches of `f`, `g`, and `h` relative to the variable `v` — `(v-then f v)` is `(then f)` if `v` = `(test f)` and `f` otherwise. The results of the recursive calls are then combined into a new decision node unless they are `bdd=`.

The statement of correctness for `ite-spec` is the theorem `ite-spec-is-correct` in Figure 3. The fact that `ite-spec` returns `robdds` is easily shown because (1) `top-var` is strictly decreasing in recursive calls, and (2) the then and else branches are guaranteed not to be `bdd=`. The proof that `prop-ev` of `(ite-spec f g h)` reduces to `prop-if` of `f`, `g`, `h` arises from a case split on `(prop-look (top-var f g h) a)` and the fact that `bdd=` is a congruence of `prop-ev`. Overall, once it was determined what properties were needed, it was a straightforward ACL2 exercise to prove the necessary theorems about the simplified BDD operations. Proving these theorems about the stobj-based implementation directly would have required a substantially greater amount of effort. We complete the presentation of the simplified operations with the statement of some reductions of `ite-spec` in Figure 4 which were used to optimize the evaluation of the implementation function `ite-bdd` defined in the next section.

3 Stobj-based BDD Implementation

The BDD manager implementation we present follows the procedures outlined in [1]. The following four functions are "exported" from the BDD manager book:

1. `((var-bdd n bdd-mgr) -> (bdd bdd-mgr))` returns a single bdd node, which splits on the variable `n`, and the updated `bdd-mgr`.

```
(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))))
```

Figure 4: Theorems about `ite-spec` optimizations

2. `((eql-bdd f g) -> boolean)` takes two bdds `f, g` and returns `T` iff the bdds `f` and `g` are semantically equivalent.

3. `((ite-bdd f g h bdd-mgr) -> (bdd bdd-mgr))` performs the ite operation on the bdds `f`, `g`, and `h` and returns the resulting bdd along-with the updated `bdd-mgr`.

4. `((free-bdd keep bdd-mgr) -> (lst bdd-mgr))` clears the `bdd-mgr` tables and then rebuilds the bdds in the list `keep`. `free-bdd` returns the list of rebuilt bdds and the updated `bdd-mgr`. This function supports both the initialization of the BDD manager, and the clearing of references to BDD nodes so the space for these nodes can be garbage collected by the Lisp runtime environment.

The `bdd-mgr` parameter in these functions is the single-threaded object defined by the `defstobj` form in Figure 5. The `bdd-mgr` has three fields: `uniq-tbl`, `rslt-tbl`, and `next-id`. The `next-id` field stores a counter which is incremented every time a new BDD node is allocated and is used to provide each new BDD node with a unique `tag`. The field `uniq-tbl` is a hash table used to uniquely construct BDD nodes. Every entry, with address I, in `uniq-tbl` is a list of BDD nodes which hash to the address I. The field `rslt-tbl` is used to cache previous results of if-then-else constructions. Every entry, with address I, in `rslt-tbl` is either `nil` or a list of four values `(f g h rslt)` where `(f g h)` hashes to address I and `(bdd= rslt (ite-spec f g h))`.

The functions in Figure 6 define the optimized stobj-based versions of the corresponding simplified functions in Figure 3. The function `eql-bdd` reduces to `iff` when comparing atoms and `tag`-equality when comparing conses. As we mentioned before, `tag`s are assigned uniquely to every BDD node which is constructed. This is maintained with the function `get-unique` which takes the values `test`, `then`, and `else` and looks them up in the `uniq-tbl` to determine if a previously-built BDD node matches these values. In a sense, `get-unique` is the stobj-based version of `dn`. Assuming constant-time hashing, `get-unique` operates in constant-time. The correspondence of `var-bdd` with `var-spec` should now be apparent and so we turn our attention to `ite-bdd`. In the definition of `ite-bdd`, we use a macro `seq` to define a sequence of bindings where the bindings may involve one or possibly more than one variable. `(seq (`b_0 b_1 `...` b_n`)` *value*`)` expands to a nest of alternating

```
(defstobj bdd-mgr
  (uniq-tbl :type (array (satisfies bdd-tr-listp) (4096))
            :initially nil)
  (rslt-tbl :type (array (satisfies ite-resultp?) (4096))
            :initially nil)
  (next-id  :type (integer 1 *)
            :initially 1))

;; the array sizes 4096 above are arbitrary and could be replaced
;; with any power-of-2 (with a recertification of the bdd books)
```

Figure 5: BDD Manager single threaded object

lets and mv-lets when the first element in each b_i is a symbol or list of symbols, respectively. For instance:

```
(seq (((u v) (mv 1 2))
      (x (+ 3 4))
      ((y z) (mv 5 6)))
     (+ x y z))
---- macro-expands to ----
(mv-let (u v) (mv 1 2)
  (let ((x (+ 3 4)))
    (mv-let (y z) (mv 5 6)
      (+ x y z))))
```

seq is particularly useful in writing functions which update stobjs since many functions which use stobjs will return multiple-values and consist of sequences of updates to the stobjs.

The function ite-bdd is structurally similar to the function ite-spec. ite-bdd first determines if f is an atom and returns g or h appropriately if this is the case. Note that ACL2 will require ite-bdd to have a consistent signature, so even in the cases when we do not update bdd-mgr, we still must return it. The next four tests in ite-bdd are optimizations corresponding to the reductions of ite-spec presented in Figure 4. If none of these cases apply, then ite-bdd will look for a match in the result cache. If a matching entry is found, then the result field of this entry is returned. If an entry is not found, then we perform recursive calls of ite-bdd (similar to the recursive calls in ite-spec), we use get-unique to construct the decision node uniquely, add the result of this call to the result cache, and then finally return the result.

The necessary correctness lemmas for eql-bdd and ite-bdd are defined in Figure 8. It is important to note that these theorems are about functions defined on the stobj bdd-mgr but the theorems themselves make no reference to the bdd-mgr. Indeed, one of the main goals in adding stobjs to ACL2[10] was that the logic should remain unaffected. This is the case since the declaration and use of stobjs enforces sufficient restrictions to ensure a correspondence between the stobj-based functions and their corresponding applicative semantics. This feature is important because it allows the ACL2 user to define elegant functions for definitions used in proofs and efficient functions for definitions which are executed. We use the variable name bmr in place of bdd-mgr in theorems. This name switch is simply for presentation clarity, since no special distinction is made for the name bdd-mgr in the logic.

The predicate bdd-mgr-inv defines the notion of a “well-formed” bdd-mgr and should be an invariant which is preserved by every operation which updates the bdd-mgr stobj. bdd-mgr-inv is defined in Figure 7. We note that while bdd-mgr-inv is an invariant defining the properties of a well-formed bdd-mgr in the logic, it could not be called on the stobj bdd-mgr itself. We chose this

```
(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))  ;; reduction-1
        ((eql-bdd g h)            (mv g bdd-mgr))  ;; reduction-2
        ((eql-bdd f g) (ite-bdd f T   h bdd-mgr))  ;; reduction-3
        ((eql-bdd f h) (ite-bdd f g nil bdd-mgr))  ;; reduction-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))))))
```

Figure 6: Definition of bdd operations

```
(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))))
```

Figure 7: Invariant definitions

approach to allow the use of functions which recur down lists to replace iteration over elements in stobj arrays. It is easier to prove theorems about functions which recur down lists than functions which recur through arrays and the flexibility of stobjs in the ACL2 logic affords us this choice. The invariant `bdd-mgr-inv` is comprised of the following conjuncts:

1. `(integerp (next-tag bmr))` – `next-tag` is the same function as `next-id`. This conjunct ensures that `(1+ (next-tag bmr))` does not equal any number less than or equal to `(next-tag bmr)`.

2. `(consesp uniq-lst)` – Every bdd node in the `uniq-tbl` is a `cons`. This is essentially a type predicate.

3. `(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.

4. `(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`.

5. `(no-dup-nodes uniq-lst)` – No two nodes in the `uniq-tbl` are `bdd=`. This ensures the uniqueness of nodes (upto `bdd=`) in the `bdd-mgr`.

6. `(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`[2]. Any operation performed on BDDs will require that the BDDs satisfy `in-uniq-tbl` in order to ensure their unique construction. We need the property that the `uniq-tbl` is contained in itself since for various inputs, `get-unique` may return any node currently in the `uniq-tbl` and this node must satisfy `in-uniq-tbl`.

7. `(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.

[2]A node `f` is *embedded* in a `tbl` iff when `f` is a `cons`, then `f` is a member of `tbl` and `(then f)` and `(else f)` are both embedded in `tbl` as well

```
(defthm eql-bdd-is-correct
  (implies (and (uniq-tbl-inv bmr)  ;; implied by (bdd-mgr-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
```

Figure 8: Correctness theorems for `bdd` functions

8. `(rslts-contained rslt-lst uniq-lst)` – Every entry in the result cache is a list `(f g h rslt)` of bdd nodes each of which satisfy `in-uniq-tbl`.

9. `(ite-results (rslt-tbl bmr))` – Every entry in the result cache is a list `(f g h rslt)` where `(bdd= rslt (ite-spec f g h))`.

The theorem `eql-bdd-is-correct` in Figure 8 states the correspondence between `eql-bdd` and `eql-spec`. The theorem is proven in the forward direction because of the `in-uniq-tbl` assumptions and because `(no-dup-tags uniq-lst)` ensures that no two nodes in the `uniq-tbl` have the same `tag`. It is proven in the reverse direction because `(no-dup-nodes uniq-lst)` ensures that no two nodes are `bdd=`.

The theorems `ite-bdd-preserves-in-uniq-tbl` and `ite-bdd-is-correct` state the correctness of the function `ite-bdd`. For every function which updates the `bdd-mgr`, we must prove a theorem similar to `ite-bdd-preserves-in-uniq-tbl`. This is necessary since many BDD nodes may be created, compared and combined at different points of the evolution of the `bdd-mgr`. Thus, an update of the `bdd-mgr` stobj should not invalidate any existing BDD node. The one exception to this is the function `free-bdd` which clears the `bdd-mgr` and only ensures that the list of BDDs it returns satisfy `in-uniq-tbl`. Once similar theorems are proven about the functions `get-unique` and `set-result`, the theorem `ite-bdd-preserves-in-uniq-tbl` is easily verified.

The theorem `ite-bdd-is-correct` cannot be proven directly; it has to be broken into two steps. In the first step, we must prove that the result satisfies `in-uniq-tbl` and that the predicate `uniq-tbl-inv` holds. We must prove these properties first in order to make use of `eql-bdd-is--correct`, which is necessary in order to show that the evaluations of `ite-spec` and `ite-bdd` correspond. Once this first step is shown, we can then prove the `bdd=`-equivalence between `ite-spec` and `ite-bdd`. This also requires that we prove in this second step that the remaining conjunct `(ite-results (rslt-tbl bmr))` is valid since we may at any point return an entry in the result cache or set an entry in the result cache. Once the first step of `ite-bdd-is-correct` is proven, the

```
(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
  (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)))))
```

Figure 9: `bdd-sat?` satisfiability checker and correctness

second step follows from the definitions of `ite-spec` and `ite-bdd` and the reductions of `ite-spec` listed in Figure 4.

We now finally prove that our BDD manager implementation can be used in defining a valid satisfiability checker. Our satisfiability checker is the function `bdd-sat?` in Figure 9 and the statement of correctness is the theorem `bdd-sat?-is-sat-checker`. The function `bdd-sat?` first clears the `bdd-mgr`, then builds the BDD for the given term, and finally tests the resulting term against `nil`. The function `clear-bdd` is actually a call to the function `free-bdd`:

```
(defun clear-bdd (bdd-mgr)
  (mv-let (temp bdd-mgr)
      (free-bdd () bdd-mgr)
    (declare (ignore temp))
    bdd-mgr))
```

The relevant property about `clear-bdd` is the following theorem:

```
(defthm clear-bdd-is-correct
  (implies (bdd-mgrp bmr)
           (bdd-mgr-inv (clear-bdd bmr))))
```

The function `bdd-mgrp` is the type predicate for the `bdd-mgr` stobj which ACL2 automatically generates from the type information about the fields. In proving the guards of the functions which operate on the `bdd-mgr`, we proved that they preserve this predicate during their execution. Thus, at any point during a run of ACL2, the `bdd-mgr` stobj will satisfy `bdd-mgrp`[3] and further, `(clear-bdd bdd-mgr)` will ensure `bdd-mgr-inv`. We note that the above satisfiability checker is not a general application of the function `free-bdd`. In light of this, we proved additional theorems about `free-bdd` to ensure that the BDD list it returns is equivalent to the BDD list it was given and that every BDD in the returned list satisfies `in-uniq-tbl`.

The function `term->bdd` transforms an arbitrary propositional term into a BDD. The important property of this transformation is the theorem `term->bdd-is-correct` which states that the resulting BDD has the same valuations as the original term under `prop-ev`. We also needed to prove that `term->bdd` preserved the `bdd-mgr-inv` and returns a BDD that satisfies `in-uniq-tbl`, but it should be apparent by now that any function which uses the `bdd-mgr` operations correctly, will necessarily satisfy these properties. Combining the theorem `term->bdd-is-correct` with the canonical form property of BDDs, we know that the original term is satisfiable if and only if the result of `term->bdd` is non-`nil`. For our witness function `sat-witness`, we simply construct the BDD for the term and then call `robdd-witness`:

```
(defun sat-witness (f bdd-mgr)
  (seq ((bdd-mgr (clear-bdd bdd-mgr))
        ((f-bdd bdd-mgr) (term->bdd f bdd-mgr)))
       (mv (robdd-witness f-bdd nil) bdd-mgr)))
```

4 Extensions and Experiments

We conclude with a discussion of possible extensions to the work presented in this paper. In short, the work documented in this paper could be considered a first cut at defining and analyzing an efficient BDD manager in ACL2. There are many directions in which this work could be extended and we consider some of these directions here.

Common Lisp Optimizations. Some common techniques for optimizing Lisp functions could be employed to improve the execution speed of our BDD implementation. To begin with, several functions that we defined could have instead been defined as macros to avoid function calls. The difficulty with the use of macros is the inability to disable their definition. For instance, if the function `eql-bdd` were instead defined as a macro, the theorems involving `ite-bdd` would have experienced a significant case explosion. Defining `eql-bdd` as a function allowed its definition to be hidden once its correspondence with `bdd=` had been shown. An alternative approach (suggested by Matt Kaufmann) is to introduce another book of definitions syntactically identical to the current implementation except that several key functions would be redefined as macros. Presumably, we could then prove that the functions in this new book were equivalent to the current implementation (assuming ACL2 could handle the large terms after expansion). Additional efficiency could be achieved through the addition of type declarations. Type declarations are especially useful in optimizing integer operations whose inputs and outputs are guaranteed to be fixnums. Unfortunately, the "logical" definition of BDDs is not restricted to any bound in size and as such we would have to change the functionality of the current implementation to enforce fixnum bounds.

Memory Management. As it stands now, for every BDD node which is created, we allocate 4 conses – 3 conses for the BDD node itself and 1 cons for the chain in the `uniq-tbl`. There are

[3]Assuming only the refined BDD manager operations are allowed to update the `bdd-mgr` stobj

time and space costs to each of these conses. In GNU Common Lisp for instance, a `cons` cell takes three 32-bit words in memory. Thus, a BDD node takes up at least 12 words in memory. Further, there is overhead in allocating and accessing `car` and `cdr` fields of conses. If millions of BDD nodes are created (and in many potential applications this is indeed the case) then the overhead of all the consing will dramatically effect the overall efficiency of the Lisp runtime environment. One solution to this problem is to manage our own node allocation and access in a large array field of a stobj. We could define each BDD as an index into the table where for each index we associate four numbers $\langle test, then, else, next\rangle$ where $test$ is a variable identifier, $then, else$ are BDD indexes, and $next$ is the index of the next BDD node in the given unique chain; the BDD index itself can be used as the tag for the node. This approach would require no consing and is potentially much more efficient – although lookup of BDD nodes would require stobj accesses which have some overhead. Unfortunately (at least for this problem) most common lisp environments enforce a limit on the size of any array. In Franz Allegro Common Lisp, this limit is about 16 million entries. This would limit the number of BDD nodes stored in a single array to around 4 million. Depending on the application involved, this number may not be sufficient and some alternative approach, such as using multiple arrays or using arrays and conses, would be required.

Variable Reordering. It is common knowledge that the number of BDDs required to represent certain propositional terms is very sensitive to the ordering of the propositional variables. For instance, representing the equality of two bit vectors of size N may require $\approx 3N$ BDD nodes for the best variable ordering and $\approx 2^N$ BDD nodes for the worst variable ordering. One technique that many existing BDD managers use to combat this problem is to dynamically reorder the variables using a variety of heuristics. One elegant approach is to implement a procedure called "sifting" which simply checks for each adjacent pair of variables (v,$v+1$) to see if flipping their order reduces the total number of nodes. The idea is that if you repeat this procedure, you eventually will obtain a good ordering on the variables. Dynamic variable reordering is a non-trivial extension and requires some extra cost in indirect node addressing and maintaining chains of nodes for common variable identifiers. Additionally, the problems we are targeting with BDDs in an ACL2 environment will rarely (if ever) represent arithmetic or equality operations at the bit-level and as such, the benefits of supporting reordering are not as apparent in this context.

Primitive Complement. Another common BDD optimization is to introduce a primitive complementation operator which is set and cleared using a single bit in a BDD address or index (e.g. negative BDD indexes are complements of the corresponding positive index). Using this tagging, computing the complement of a BDD is a very fast constant-time operation. Further, using the complement primitive affords a greater amount of normalization in BDD representations, which in turn leads to fewer allocated nodes and more efficient use of the result cache.

Partitioned Image Computation. A common application of BDDs is for so-called symbolic model checkers[2]. Model checking is a procedure for checking that some system satisfies a formula from some temporal logic (often CTL, FairCTL, or LTL). Model checking procedures verify this satisfaction by exploring the reachable states of the system to be checked. BDDs are often used in symbolic model checkers as the compact representation of a set of states. A common operation performed in model checking is the computation of a set of next-states given a current state set and a transition relation, termed an image computation. Where a state set may be defined on N propositional variables, a straightforward encoding of a transition relation requires $2N$ propositional variables (current state var.s and next state var.s). This straightforward encoding of a transition relation can be prohibitively expensive to build and use in practice and so an alternative procedure for performing image computations is desired. One alternative is to define the transition relation with N BDDs – one for each next state variable x' – each of which is defined on N current state variables and defines the set of states for which x' is 1. This approach requires a special procedure to perform image computations but often avoids the computational overhead of the straightforward encoding. Although we do not document this extension here, we have added this procedure to the BDD manager defined in this paper. In a future paper we will cover this extension.

ACL2 Term-Level BDDs. The BDDs we defined in this paper used propositional variables for the test fields of the `if`-expressions. In a general ACL2 application of BDDs, we would like to use arbitrary ACL2 terms (i.e. either a quoted constant, variable symbol, or function application on ACL2 terms) in the test positions of the `if`-expressions. Indeed, ACL2s current BDD package supports arbitrary terms at the test and leaves of the `if`-expressions. For our purposes, we would like to use arbitrary ACL2 terms at test positions while ensuring the BDD computed is a canonical form for the set of interpretations of the free variables of these terms. Unfortunately, these terms will likely involve common free variables and thus, are not independently `nil` or non-`nil`. Extending BDDs to general terms in this manner will likely require additional proof requirements of the user and methods for achieving this interaction is one of our current areas of research interest.

Experiments. Finally, we present some experimental results to facilitate the performance comparison of the BDD manager presented in this paper with two versions written and compiled in C. Unfortunately, the version of the BDD manager presented in this paper is very inefficient for numerous reasons. So in order to make a meaningful comparison of what can be done in ACL2 using stobjs and the various extensions presented above, we wrote an optimized version of the BDD manager which included the Common Lisp Optimizations, Memory Management, and Primitive Complement extensions mentioned above. It is important to note that as of the time of this writing, this optimized BDD manager has not been verified. The functions defining the optimized manager have been admitted to the logic, and have had their guards verified. It is the author's intention in the near future to complete a proof of this optimized BDD manager – similar in structure to the one outlined in this paper. We compare this optimized implementation in ACL2 with a straightforward translation of this manager (by hand) to C and with the CUDD[12] package, which is also written in C. The ACL2 BDD manager was compiled using Franz[5] Allegro Common Lisp, and the two C managers were compiled using the Gnu Compiler Collection(GCC) with optimization level three[6]. These three managers were run on various instances of three different problems. The first problem is Urquhart's U-formula which was used as a benchmark in [13]. For N propositional variables $x_1, .., x_N$, the U-formula is ($\Leftrightarrow$ is propositional equivalence):

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

We also tested the managers using the problem of multiplying of two size-N bit-vectors, and some random tests where the parameter N is used as a seed for a random BDD generator. The tests were performed on an UltraSparc workstation under Solaris, with Franz version 5.0.1 and GCC version 2.7.3.2. The following table presents the execution times in seconds, with "ACL2" denoting the optimized BDD manager in ACL2, "GCC" denoting the translation by hand of the ACL2 BDD manager, and "CUDD" denoting the CUDD package compiled with GCC as well.

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

Generally speaking, GCC and CUDD performed similarly with a few exceptions. CUDD has some overhead for BDD nodes due to its support for garbage collection using reference counts and dynamic variable reordering. CUDD also has support for reallocating result caches and unique tables on demand. Unfortunately, stobj arrays in ACL2 must have a fixed allocation which does not allow the user to size them depending on the demands of a particular problem. Since the GCC manager was a straightforward translation of the ACL2 manager, it shares this property of fixed allocation which may explain the difference in growth (2^N vs. 4^N) in the multiplication example since this problem exhibits a great deal of result sharing. It is also worth noting that for CUDD, we disabled dynamic variable reordering since it didn't help in any of these examples. In actual systems, though, dynamic variable reordering can be crucial if a bad variable ordering is provided and it would be easy to construct relatively small examples with poor initial variable orders where CUDD is exponentially faster than our implementation. Still, the limited results we present here are encouraging and demonstrate how close optimized ACL2 using stobjs can get to C-like efficiency.

References

[1] K. Brace, R. Rudell, and R. Bryant. Efficient Implementation of a BDD package, *Proceedings of the ACM/IEEE Design Automation Conference*, 1990.

[2] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} states and beyond," in *IEEE Symposium on Logic in Computer Science*, 1988.

[3] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers*, August, 1986.

[4] R. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. Tech. Report CMU-CS-92-160, School of Computer Science, Carnegie Mellon University, 1992.

[5] Franz Allegro Common Lisp home page: http://www.franz.com/

[6] Gnu Compiler Collection home page: http://gcc.gnu.org/

[7] D. Greve, M. Wilding, and D. Hardin. High-Speed, Analyzable Simulators, in *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June, 2000.

[8] J. Harrison. Binary Decision Diagrams as a HOL derived rule, in *The Computer Journal.* volume 38, 1995.

[9] S. Minato. Binary Decision Diagrams and Applications for VLSI CAD. Kluwer Academic Publishers, 1996.

[10] J. Moore and R. Boyer. Single-Threaded Objects in ACL2. available from the ACL2 publications web page: http://www.cs.utexas.edu/users/moore/publications/acl2-papers.html

[11] P. Manolios. Mu-calculus Model Checking, in *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Publishers, June, 2000.

[12] F. Somenzi, et. al. CUDD: CU Decision Diagram Package. Available at the CUDD web page: http://vlsi.colorado.edu/ fabio/CUDD/

[13] K. Verma and J. Goubault-Larrecq. Reflecting BDDs in Coq. INRIA Research Report 3859, January, 2000.