Reflecting BDDs in Coq

Kumar Neeraj Verma[1], Jean Goubault-Larrecq[1], Sanjiva Prasad[2], and S. Arun-Kumar[2]

[1] GIE Dyade and INRIA Rocquencourt
[2] IIT Delhi

Abstract. We describe an implementation and a proof of correctness of binary decision diagrams (BDDs), completely formalized in Coq. This allows us to run BDD-based algorithms inside Coq and paves the way for a smooth integration of symbolic model checking in the Coq proof assistant by using reflection. It also gives us, by Coq's extraction mechanism, certified BDD algorithms implemented in Caml. We also implement and prove correct a garbage collector for our implementation of BDDs inside Coq. Our experiments show that this approach works in practice, and is able to solve both relatively hard propositional problems and actual industrial hardware verification tasks.

1 Introduction

Binary Decision Diagrams (BDDs for short) [9] are a compact and canonical representation of propositional formulae up to propositional equivalence, or equivalently of Boolean functions. BDDs and related data structures are at the heart of modern automated verification systems, based on model-checking [24] or on direct evaluation of observable equivalence between finite-state machines [10]. These techniques have enabled solving huge verification problems automatically. On the other hand, proof assistants like Coq [3], Lego [23] or HOL [15] are *expressive*, in that they can address a vast amount of rich mathematical theory (not just finite-state machines or Boolean functions). The expressiveness provided by such systems makes them eminently suitable for diverse verification tasks, such as expressing modular properties of hardware circuits [22].

Proof assistants are also *safe*, in that they rest on firm logical foundations, for which consistency proofs are known. While implementations of proof assistants might of course be faulty, some design principles help limit this to an absolute minimum. In HOL, the main design principle, inherited from LCF [16], is that every deduction, whatever its size may be, must be justified in terms of a a small number of indisputable elementary logical principles. In Coq and Lego, every proof-search phase is followed by an independent proof-checking phase, so that buggy proof-search algorithms cannot fool the system into believing logically incorrect arguments.

However, proof assistants are not automatic, even when it comes to specialized domains like Boolean functions. In particular, proof assistants today cannot perform automatic model-checking (PVS [27] is a notable exception, see below.)

To combine the expressiveness and safety of proof assistants with model-checking capabilities, there have been proposals to integrate BDDs with proof assistants (some are discussed below), which may involve certain trade-offs. If the proof assistant is to remain safe, we cannot just force it to rely on the results provided by an external model-checker. We might instrument an external model-checker so that it returns an actual proof, which the given proof assistant will then be able to check. But model-checkers actually enumerate state spaces of astronomical sizes, and mimicking this enumeration — at least naively, see related work below — with proof rules inside the proof assistant would be foolish.

One remedy to this model-checker (automatic, fast) vs proof assistant (expressive, safe) antinomy is to use *reflection* [32, 7, 1, 4, 6], which in this context can roughly be thought of as "replacing proofs in the logic by computations" (see Section 3). Reflection is particularly applicable in Coq, since Coq is based on the calculus of inductive constructions, a logic which is essentially a typed lambda-calculus, a quintessential programming language.

In this paper we describe an implementation of BDDs in Coq using reflection. We model BDDs, implement the basic logical operations on them, and prove their correctness, all within Coq. This provides a formal proof of the BDD algorithms, and also makes available these techniques within the proof assistant. Our BDD implementation employs garbage collection. We describe how we implement a provably correct garbage collector for our BDD implementation. We provide experimental results about our BDD implementation on relatively hard propositional problems as well as industrial hardware verification benchmarks.

We must clarify that the aim of this work is not to improve on current verification technology, as far as efficiency or the size of systems verified is concerned. Our goal is to produce certified model-checkers, and also reflected model-checkers that could work as subsystems of Coq. Given that the Common Criteria (CC) certification [26] requires the use of certified formal methods at evaluation levels 5 and higher, our work may be viewed as enriching the set of tools available to people and industries engaged in the CC certification process to include proved BDD techniques, running at an acceptable speed. It has to be noted that there is also increasing interest in independent proof-checking of machine-checked proofs: here, Coq is a particularly interesting system to work with, since its core logic is so small that stand-alone, independent proof-checkers for it are not too hard to implement. In fact, such a proof-checker exists that has even been formally certified in Coq [2] (augmented with a strong normalization axiom to get around Gödel's second incompleteness theorem).

The plan of the paper is as follows. We review some related work in the rest of the introduction, then give short descriptions of BDDs in Section 2, and of Coq in Section 3. Sections 4 and 5 are the technical core of this paper: in Section 4 we describe how BDDs are not only modeled but actually programmed and proved in Coq. This involves a number of difficulties; to name a few: we have to describe memory allocation, sharing, recursion over BDDs, memoization, and to prove that they all work as expected. The organization of the correctness proofs is on classical lines: we state representational invariants, which we show are maintained by the BDD algorithms; we provide the semantics of BDDs in terms of Boolean functions, with respect to which the correctness of the algorithms is proven. For the BDD implementation to be really useful in practice, we also need to address problems related to memory management. We also describe a formally proved garbage collector for our implementation. In Section 4, we only describe the assumptions about the garbage collector and show that our BDD implementation works correctly under these assumptions. The garbage collector we implement is described and then shown to satisfy these assumptions in Section 5. The garbage collector is shown to preserve representational invariants and the semantics of all BDD nodes designated as relevant. In Section 6 we report experimental results, and discuss speed and space issues. We then conclude in Section 7.

The complete Coq code and proofs can be found at `http://www.dyade.fr/fr/actions/vip/bdd.tgz`.

Related work.

Closely related to our work is John Harrison's interfacing of a BDD library with the HOL prover [19]. The author's goal was to solve the validity problem for propositional logic formulae rather than to perform model-checking. Reflection cannot be employed in HOL, since its logical language does not contain a programming sub-language. To be precise, it does contain a λ-calculus with $\beta\eta$-equality, but this equality is not implemented as a reduction process as in Coq, but through the use of axioms defining it, which then have to be invoked explicitly. Therefore, the BDD library must log every BDD reduction step and translate each as a proper HOL inference, which HOL will then check. This logging process is difficult to implement, produces huge logs, and it is necessary to use a few tricks in the HOL implementation to keep the HOL proof-checking phase reasonably efficient.

There have also been several publications on integrating model checking with theorem proving. Indeed, this is one of the main aims of PVS [27]; but the internal model-checker of PVS is itself not checked formally, for now at least. Yu and Luo have built a model checker LegoMC [33] which returns a proof term expressing why it believes the given formula to hold on the given program. Safety is achieved through Lego checking the latter proof-term. However such methods do not seem to scale up to complex problems, since they are not symbolic checking methods like those based on BDDs.

Gordon and his colleagues have combined the HOL proof assistant with an external BDD package [13, 14], allowing the HOL prover to accept results from the external BDD package, without any rechecking by HOL. While this approach is likely to work faster, its safety is dependent on the reliability of the external BDD package as well as the mechanisms for communication between HOL and the BDD package. Our approach involves more work but is completely safe.

There have also been several works on verifying garbage collectors inside proof assistants. Goguen *et al.* [11] model memory as a directed graph, define basic operations that a mutator may apply to memory, then show that adding a garbage collector (abstracted as Dijkstra *et al*'s tricolor marking collector) gives a bisimilar, hence behaviorally equivalent, process. Their high-level view does not take into account any particular algorithmic detail, whereas we implement actual code that is part of a BDD implementation running in Coq and show that our implementation works as expected in the presence of this garbage collector. Moreover, they verify a garbage-collector for a fairly classical memory architecture, which does not include persistent hash-tables that should be purged of freed elements. So not only do we verify actual code, we also prove a more complex garbage collection architecture. This last aspect is also a difference between our work and others such as [12, 21, 25, 29].

2 Binary Decision Diagrams

BDDs represent a propositional logic formula (built from variables and the connectives $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$) as graphs. It is based on Shannon's decomposition of any formula F as $(A \Rightarrow F[A := \mathbf{1}]) \wedge (\neg A \Rightarrow F[A := \mathbf{0}])$, where $F[A := G]$ denotes substitution of G for A in F, $\mathbf{1}$ is `true` and $\mathbf{0}$ is `false`. We also use the convenient notation $F = A \longrightarrow F[A := \mathbf{1}]; F[A := \mathbf{0}]$. By choosing a variable A occurring in F, decomposing with respect to it, and recursively continuing the process for $F[A := \mathbf{1}]$ and $F[A := \mathbf{0}]$, we get a binary tree (a *Shannon tree*) whose internal nodes have variables and leaf nodes are $\mathbf{1}$ and $\mathbf{0}$.

BDDs are *shared*, *reduced* and *ordered* versions of Shannon trees. *Sharing* means that all isomorphic subtrees are stored at the same address in memory. This gives us directed acyclic graphs instead of trees. It is also the main reason why BDDs are very small in many applications. Sharing is accomplished by having a sharing map which remembers all the BDDs that have been previously constructed. *Reducing* means that a BDD node with identical children represents a redundant comparison and can be replaced by a child node. *Ordering* means that we fix some ordering on the variables and have all the variables along any path in a BDD occur in that order. These conditions ensure that BDDs are canonical forms for propositional formulae up to propositional equivalence [9].

One reason why BDDs are interesting is that they are usually compact. Furthermore, they are easy to work with, at least in principle: all the usual logical operations are easily definable by recursive descent on BDDs. Negation, for example, is defined by $\neg\mathbf{1} =_{\mathrm{df}} \mathbf{0}$, $\neg\mathbf{0} =_{\mathrm{df}} \mathbf{1}$, $\neg(A \longrightarrow F; G) =_{\mathrm{df}} A \longrightarrow \neg F; \neg G$, and any other Boolean operation $\oplus$ (ranging over $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$, etc.) is defined as $(A \longrightarrow F_1; G_1) \oplus (A \longrightarrow F_2; G_2) =_{\mathrm{df}} A \longrightarrow (F_1 \oplus F_2); (G_1 \oplus G_2)$, with the usual definitions on $\mathbf{1}$ and $\mathbf{0}$. This is called *orthogonality* of operation $\oplus$.

By remembering in a cache the results of previous computations, negation can be computed in linear time, and binary operations in quadratic time w.r.t. the sizes of the input BDDs, a notion sometimes called *memoizing* or *tabulating*.

3 An Overview of Coq

Coq is a proof assistant based on the Calculus of Inductive Constructions (CIC), a type theory that is powerful enough to formalize most of mathematics. It properly includes higher-order intuitionistic logic, augmented with definitional mechanisms for inductively defined types, sets, propositions and relations. CIC is also a typed λ-calculus, and can therefore also be used as a programming language. For a gentle introduction, see [20]. We shall describe here the main features of Coq that we shall need later, and also what reflection means in our setting.

The *sorts* of CIC are Prop, the sort of all propositions; Set, the sort of all specifications, programs and data types; and Type_i, $i \in \mathbb{N}$, which we won't need. The typing rules for sorts include $\mathrm{Prop} : \mathrm{Type}_0$, $\mathrm{Set} : \mathrm{Type}_0$, $\mathrm{Type}_i : \mathrm{Type}_{i+1}$.

What it means for Prop to be the sort of propositions is that any object $F : \mathrm{Prop}$ (read: F of type Prop) denotes a proposition, i.e., a formula. If F and G are propositions, F `->` G is a proposition (read: "F implies G"), and (`x` :T)G is a proposition, for any type T (read: "for every `x` of type T, G"). In turn, any object $\pi : F$ is a *proof* π of F. A formula is considered proved in Coq whenever we have succeeded to find a proof of it. Proofs are written, at least internally, as λ-terms, but we shall be content to know that we can produce them with the help of *tactics*, allowing one to write proofs by reducing the goal to subgoals, and eventually to immediate, basic inferences.

Similarly, any object t : Set denotes a data type, like the data type `nat` of natural numbers, or the type `nat -> nat` of all functions from naturals to naturals. Data types can, and will usually be, defined inductively. For instance, the standard definition of the type `nat` of natural numbers in Coq reads:

```
Inductive nat : Set :=
    O : nat
  | S : nat -> nat.
```

which means that `nat` is of type Set, hence is a data type, that `O` (zero) is a natural number, that `S` is a function from naturals to naturals (successor), and that every natural number must be of the form (`S` (`S` ...(`S` `O`)...)) (induction).

Then, if t : Set, and $p : t$, we say that p is a *program* of type t. Programs in Coq are purely functional programs: the language of programs is based on a λ-calculus with variables, application (`p q`) of `p` to `q` (in general, ($\mathtt{p}\ \mathtt{q_1}\ \ldots\ \mathtt{q_n}$) will denote the same as $(\ldots(\mathtt{p}\ \mathtt{q_1})\ldots\mathtt{q_n})$), abstraction [`x` : `t`]`p` (where `x` is a variable; this denotes the function mapping `x` of type `t` to `p`), case splits (e.g., `Cases n of O => v | (S m) => (f m) end` either returns `v` if the natural number `n` is zero (`O`) or (`f m`) if `n` is the successor of some integer `m`), and functions defined by structural recursion on their last argument. For the latter, consider the following definition in Coq's standard prelude:

```
Fixpoint plus [n:nat] : nat -> nat :=
  [m:nat] Cases n of
            O => m
          | (S p) => (S (plus p m))
          end
```

This is the standard definition of addition of natural numbers, by primitive recursion over the first argument. For soundness reasons, Coq refuses to accept any non-terminating function. In fact, Coq refuses to acknowledge any `Fixpoint` definition that is not primitive recursive, even though its termination might be obvious.

In general, propositions and programs can be mixed. We shall exploit this mixing of propositions and programs in a very limited form: when π and π' are programs (a.k.a., descriptions of data), then π=π' is a proposition, expressing that π and π' have the same value. Reflection can then be implemented as follows: given a property P : t -> `Prop`, write a program π : t -> `bool` deciding P. (Note, by the way, that `bool` is defined by `Inductive bool : Set := true : bool | false : bool.` and should be distinguished from `Prop`). Then prove the correctness lemma:

$$(x : t)\ (\pi\ x)\ \texttt{= true ->}\ (P\ x)$$

To show that P holds of some concrete data x_0 (without any free Coq variable), you can then compute (π x_0) in Coq's *programming language*. If the result is `true`, then (π x_0)=`true` holds, and the correctness lemma then allows you to conclude that (P x_0) holds in Coq's *logic*.

More concretely, assume we are interested in the validity of Boolean expressions. The type of Boolean expressions is:

```
Inductive bool_expr : Set :=
      Zero : bool_expr                        (* false *)
    | One : bool_expr                         (* true *)
    | Var : BDDvar->bool_expr                 (* propositional variables *)
    | Neg : bool_expr->bool_expr              (* negation *)
    | Or : bool_expr->bool_expr->bool_expr    (* disjunction *)
    | And : bool_expr->bool_expr->bool_expr   (* conjunction *)
    | Impl : bool_expr->bool_expr->bool_expr  (* implication *)
    | Iff : bool_expr->bool_expr->bool_expr.  (* logical equivalence *)
```

(The type `BDDvar` of variables will be described in Section 4.1.) Given an environment ρ mapping variables to truth values (in `bool`), it is easy to define the truth-value of a given Boolean expression `be` under ρ, by standard evaluation rules: `Zero` is always false, (`Var` x) is true if and only if $\rho(x)$ is, (`And` be_1 be_2) is true provided both be_1 and be_2 are, and so on. A Boolean expression is *valid* if and only if its value is true under every environment ρ.

It is often the case that we are interested in showing that a given Boolean expression `be` is valid. The standard proof of it is to list its free variables, say $x_1, \ldots, x_n$, then do a case analysis on the truth-value of x_1; in each of the two cases (x_1 true, or x_1 false), do a case analysis on x_2, and so on. In the worst case, this requires writing a proof of size $O(2^n)$, which is tedious or even infeasible for n large enough. Furthermore, even though writing this proof may in principle be automated, by writing a computer program whose role is to output all the necessary proof steps to be fed to the Coq proof engine, the sheer size of it will make its verification consume too much time and space to be feasible at all, for $n \geq 30$ at least on current machines.

Bypassing this problem can be accomplished by devising proof rules, and showing that they are sound with respect to the semantics of Boolean expressions. We then let the user, or some computer program implementing a decision procedure for propositional logic, generate the proper sequence of applications of these proof rules to establish that `be` is valid (which is possible provided these proof rules are complete), and the proof assistant will then re-check that this sequence of applications is correct. This is what Boutin [6] calls *partial reflection*. This is also what Harrison [19] implemented in HOL, using BDDs as decision procedure; we have already seen in Section 1 why this solution was not completely satisfactory.

Our solution is to use *total reflection* [6]: we write a function `is_tauto` in Coq's λ-calculus that takes a Boolean expression `be` as input and returns whether the BDD for `be` is the distinguished **1** or not. We then prove the fundamental correctness lemma `is_tauto_lemma`, which states that if `(is_tauto be)` equals `true`, then `be` is valid. (The actual definitions are slightly more elaborate, see Section 4 for the precise formulation, implementation and proof strategies.) The task of any user wishing to show that a given Boolean expression `be` is valid will then be as follows: submit this claim to Coq, which will ask for a proof; then type `Apply is_tauto_lemma`: Coq now asks for a proof of `(is_tauto be)=true`. Finally, type `Reflexivity` to instruct Coq that both sides of the equals sign in fact have the same value. Since they are not syntactically the same, Coq will start computing, and simplify both until either they are equal (in which case the proof succeeds), or until they are not and no simplification is possible (then the proof fails).

At this point, the proof is finished. It has been completely automated, it is small (the size of the generated proof-term is only that of the goal we proved plus some constant overhead, since computation steps are not recorded), and it is as safe as the rest of Coq (we never had to trust an external BDD package blindly, in particular).

4 Implementing BDDs in Coq

In this section we describe how we model BDDs in Coq and implement and prove the basic algorithms on them. We first describe the data structures used for modeling BDDs, for implementing sharing and memoization, and for garbage collection. We define the invariants of our representation, define the semantics of BDDs in terms of Boolean functions and prove the correctness of the algorithms in terms of this semantics. In this process we also prove the uniqueness of BDDs for our representation. The garbage collector is described in Section 5. Here we describe the main BDD algorithms, with the necessary assumptions about the garbage collector. Since the Coq formalization assures technical correctness of our results, our endeavor will be to convey the underlying intuition, with only a few actual Coq definitions shown. The formal definitions and details may be found in the complete Coq code and proofs.

4.1 Representation

We use a library of maps [18] to model a *state* i.e. the memory in which all the BDDs are to be stored, and also the sharing maps that support sharing of BDDs. This library contains an implementation of finite sets and maps. The type `(Map A)` consists of maps denoting finite functions from addresses (of type `ad`, which consists of binary representations of integers) to the set `A`. `(MapGet ? m a)` returns `(NONE A)` if the address `a` is not in the domain of the map `m`, or `(SOME A x)` if `m` maps `a` to the element `x` (of type `A`). (Note the use of the question mark to abbreviate a type argument, here `A`, that the Coq type-checker is able to infer by itself. In the above case, `(MapGet ? m a)` is understood by Coq as `(MapGet A m a)`.) `(NONE A)` and `(SOME A x)` are the elements of type `(option A)`. `(MapPut ? m a x)` changes the map `m` by mapping the address `a` to `x` in `m`. Equality of addresses is defined by `ad_eq`. Addresses can be converted to natural numbers (of type `nat`) by `nat_of_ad`.

```
Definition BDDstate := (Map (BDDvar * ad * ad)).
Definition BDDsharing_map := (Map (Map (Map ad))).
```

A `BDDstate` maps addresses to triples consisting of a variable and two other addresses which represent the left and right sub-BDDs. Variables and addresses are both represented by the type `ad` (i.e. `BDDvar` = `ad`). The constants `BDDzero` and `BDDone` are the addresses zero and one and are used for the leaf nodes **0** and **1** respectively. Having variables as integers gives us a natural ordering on the variables. It also makes it easy to define the sharing map by giving it the type `(Map (Map (Map ad)))` (the domain of maps can only be addresses). A sharing map represents the inverse function of a BDD state and (in curried form) maps left hand addresses to maps from right hand addresses to maps from variables to addresses. We also have maps for memoization. The memoization map for negation (of type `BDDneg_memo`, defined as `(Map ad)`) maps addresses to other addresses representing the negated BDD. Memoization maps for disjunction have the type `BDDor_memo`, defined as `(Map (Map ad))`. They map pairs of addresses to addresses.

To implement allocation of new nodes and to do garbage collection, we also keep some other information in our configuration. We have a free list (of type `BDDfree_list` defined as `(list ad)`) which contains addresses that are freed during garbage collection. We also have a counter (of type `ad`), that is an address beyond which all addresses are unused. The type `BDDconfig` consists of tuples with the six components described above. All the algorithms also take in as input a *used list* (of type `(list ad)`) which contains addresses representing BDDs which are used by the user, and is used to inform the garbage collector that these BDDs should not be freed.

4.2 Invariants

Next we define the invariants of our representation to ensure that firstly, they represent well formed BDDs (i.e. directed, acyclic graphs with leaf nodes **1** and **0**, with the reducedness and ordering condition), and secondly, the various components of the data structures are consistent with each other.

Our first invariant, (BDDbounded bs node n), states that all the nodes in the BDD rooted at node have variables less than n. It also states the fact that the BDDs are reduced and ordered:

```
Inductive BDDbounded [bs:BDDstate] : ad -> BDDvar -> Prop :=
  BDDbounded_0 : (n:BDDvar) (BDDbounded bs BDDzero n)
| BDDbounded_1 : (n:BDDvar) (BDDbounded bs BDDone n)
| BDDbounded_2 : (node:ad) (n:BDDvar) (x:BDDvar) (l,r:ad)
   (MapGet ? bs node)=(SOME ? (x, (l, r)))
   -> (BDDcompare x n)=INFERIEUR -> ~l=r -> (BDDbounded bs l x) -> (BDDbounded bs r x)
    -> (BDDbounded bs node n).
```

We shall constantly require our BDD nodes node to be *bounded* by n (in a given state bs), in the sense that they satisfy (BDDbounded bs node n). Formally, the definition above states that the set $B_{\mathtt{n}}$ of BDD nodes bounded by n is the smallest containing 0 and 1 (clauses BDDbounded_0 and BDDbounded_1), and such that if node points to (x, (l, r)) in bs, then node is in $B_{\mathtt{n}}$ provided $\mathtt{x} < \mathtt{n}$ and l and r are in $B_{\mathtt{x}}$ (ordering condition: this implies that all variables are less than n and in decreasing order along each path from the root), and provided $\mathtt{l} \neq \mathtt{r}$ (reducedness). The BDDbounded predicate makes no statement about canonicity of BDDs yet: that BDDs are canonical will follow not only from all BDDs being bounded, but also from other invariants involving the well-formedness of the state bs as well as of the sharing maps (see Lemma BDDunique_1 in Section 4.3).

Observe that we have chosen to have the variables in decreasing order on all paths. This is contrary to usual convention. However it allows us to use the fact that $<$ is well-founded on ad, so the conditions defined in BDDbounded additionally imply that any bounded BDD node is the root of a *finite* graph, which is therefore in particular also *acyclic*. Without BDDbounded, enforcing acyclicity would have been much trickier.

This choice of ordering also gives a natural induction principle for doing all the proofs related to BDDs. In most paper-proofs of BDD algorithms, induction is done either on the structure of BDDs or on their sizes. But the structure of BDDs is not apparent to Coq—at least BDDs are not just inductive trees, and instead appear to Coq as a mesh of pointers—and size must itself be defined by induction over BDDs—so a more basic induction principle is needed to define size. Enforcing BDD nodes to be bounded by some number provides us with an easy way out: we just induct on a canonical number that is strictly larger than any variable in the given BDD. More precisely, the value that we use for induction in most of the proofs is bs_var' (see below), which is one more than the variable stored at the root node of the BDD. In case of leaf nodes, it is zero, thus ensuring that bs_var' strictly decreases from a node to its children, even if the children are leaf nodes:

```
Definition bs_var' := [bs:BDDstate; node:ad]
  Cases (MapGet ? bs node) of
      NONE => ad_z         (* leaf node; return ad_z (representing zero) *)
    | (SOME (x,(l,r))) => (ad_S x)     (* ad_S is the successor function *)
  end.
```

As far as the other needed invariants are concerned, let us mention the underlying ideas, rather than display the (fairly straightforward) Coq definitions. The predicate BDDstate_OK says that there is nothing stored in the BDD state at the addresses zero and one (since they are reserved for leaf nodes), and any node containing the variable n is bounded by n + 1. (i.e. satisfies the BDDbounded predicate). Sharing is ensured by the predicate BDDsharing_OK which says that the sharing map and the BDD state represent inverse functions. The predicate BDDfree_list_OK says that the free list stores exactly those nodes between 1 and counter (excluding them) which are not in the domain of the BDD state; we also require that it does not contain duplicates, as this is needed for correct behavior. The predicate counter_OK means that the counter is strictly greater than one and there is nothing stored in the BDD state starting from this address. The predicates BDDneg_memo_OK and BDDor_memo_OK define the invariants on the memoization maps. A memoization map is OK if all entries refer to nodes that are OK, (a *node is OK* if it is in the domain of the BDD state or is a leaf node) and the interpretation of the nodes as Boolean functions (see Section 4.3) are related as expected.

We have the predicate BDDconfig_OK for BDD configurations which encapsulates the six invariants described above.

Besides that, we have the predicate used_list_OK for the *used list* that is passed to all the functions as argument, which says that the used list should only contain nodes which are OK.

4.3 Interpretation as Boolean Functions

A *Boolean function* (of type bool_fun) maps environments to Booleans and an *environment* maps variables to Booleans. bool_fun_zero and bool_fun_one are the constant Boolean functions. Extensional equality is defined by bool_fun_eq. We also have the usual operations bool_fun_and, bool_fun_or, bool_fun_neg, bool_fun_impl, bool_fun_iff, bool_fun_if which are easy to define. A node node in the BDD state bs is interpreted as a Boolean function in the expected way by the following function, which we discuss below.

```
Fixpoint bool_fun_of_BDD_1 [bs:BDDstate; node:ad; bound:nat] : bool_fun :=
  Cases bound of O => (* Error *) bool_fun_zero
    | (S bound') => Cases (MapGet ? bs node) of
       NONE => (* leaf node *) if (ad_eq node BDDzero) then bool_fun_zero
                             else bool_fun_one
     | (SOME (x,(l,r))) => (bool_fun_if x (bool_fun_of_BDD_1 bs r bound')
                                          (bool_fun_of_BDD_1 bs l bound'))
        end end.
```

We come up against a difficulty while defining this function, namely that Coq only allows recursive functions which use structural induction on the last argument—this is a design principle of Coq ensuring that Coq is strongly normalizing and consistent. We would indeed like to define bool_fun_of_BDD_1 as a function of just bs and node, by induction on the structure of the BDD stored at node. However we have no direct way of doing so since BDDs are not represented as trees (since it would have prevented us from dealing with sharing). This problem is solved by supplying an extra bound bound of type nat. This argument is decreased by one in the recursive calls to the function, which makes

Coq happy. (This is sometimes known as "Boyer's trick.") We then supply a bound which is greater than the maximum depth of recursion we expect. This bound can be very easily computed from the variable that is stored at the root node (to be precise, it is one more than the quantity `bs_var'`). The semantics function `bool_fun_of_BDD_bs` (omitted here) is defined accordingly, using the above function. We also prove a theorem stating that the value returned by this function is independent of the bound passed as argument, provided that this bound is sufficiently large. We employ a similar tack in all the other functions (negation, disjunction etc.) where we need to recurse on the structure of the BDDs.

We prove the uniqueness of BDDs for our representation in terms of the addresses where they are stored. The uniqueness lemma says that if the Boolean functions represented by two nodes in a BDD state are equal then the two nodes are equal (i.e. they are at the same address). In the statement of the lemma we have the extra argument `n` which is equal to the maximum of the `bs_var'`s of the two nodes. This then enables us to apply well founded induction on `n`.

```
Lemma BDDunique_1 : (bs:BDDstate; share:BDDsharing_map) (BDDstate_OK bs)
  -> (BDDsharing_OK bs share) -> (n:nat)(node1,node2:ad)
  n=(max (nat_of_ad (bs_var' bs node1)) (nat_of_ad (bs_var' bs node2)))
  -> (node_OK bs node1) -> (node_OK bs node2)
  -> (bool_fun_eq (bool_fun_of_BDD_bs bs node1) (bool_fun_of_BDD_bs bs node2))
  ->node1=node2.
```

The proof takes around 500 steps, and is along expected lines.

Outline of proof: The cases where both `node1` and `node2` are leaf nodes are easy. If `node1` is a leaf node, then `node2` cannot be a non-leaf node because then both its children would have the same semantics, and consequently would be equal (from induction hypothesis,) violating the fact that BDDs are reduced (from the conditions in the `BDDbounded` predicate.) In case both `node1` and `node2` are non-leaf nodes, we first show that they contain the same variable, which requires us to separately prove a lemma stating that the Boolean function represented by a node in a configuration is independent of the variables greater than the variable at the root node. We then use the induction hypothesis to derive that the left children are equal and the right children are equal. Finally we apply the conditions in the `BDDsharing_OK` predicate to show that the two nodes are equal. □

4.4 Assumptions about the Garbage Collector

As mentioned earlier, our implementation employs garbage collection. We factor the verification into two parts – proving the BDD algorithms are correct assuming the garbage collector satisfies certain specifications, and then showing (in Section 5) that the garbage collector we implement satisfies these specifications. This factoring also gives us the flexibility to change the garbage collector (like changing the conditions when memory should be freed) at some later time independently of the BDD algorithms.

The specifications of the garbage collection function are: it takes in as input a BDD configuration and a used list and returns a new configuration. Its behavior is specified using the predicate gc_OK, which says that the new configuration returned by the garbage collector is OK, the contents of all nodes reachable from the used list are preserved and no new nodes are added into the configuration.

4.5 Memory Allocation

The main function responsible for changing the store is BDDmake. All the BDD algorithms modify the store by calling this function. It takes in as input a variable x and two BDD nodes l and r and returns a node representing the Boolean function "if x then *bfr* else *bfl*", where *bfr* and *bfl* are the Boolean functions represented by r and l respectively. It first checks whether such a node is already present in the configuration, and if not then it calls the function to allocate a new node. The allocation function, as well as all the other BDD algorithms are parameterized by a garbage collection function gc. The allocator first calls gc (which may or may not choose to free memory depending on the requirements), then allocates a node from the free list, if available, or otherwise it allocates the node at the address pointed to by the counter and the counter is incremented by one.

4.6 BDD Algorithms

We implement negation and disjunction as the basic operations on which the rest of the operations (conjunction, implication and double implication) are based. Thus we have two memoization tables in the configuration. The algorithms work by simple recursion on the structure of the BDDs. However the extra complexity is because of the fact that these algorithms are not purely functional in nature and need to modify the store. Each recursive call to the function returns a new configuration. Finally we call BDDmake which returns another configuration.

We use the following function for negation. As described in Section 4.3, recursion on the structure of BDDs is accomplished by having an extra argument of bound type nat to bind the maximum depth of recursion. gc is the garbage collection function, ul is the used list and node is the node whose negation is to be computed. The function returns a new configuration and a new node in it representing the negated BDD. The computed result is memoized by the function BDDneg_memo_put.

```
Fixpoint BDDneg_1 [gc:(BDDconfig->(list ad)->BDDconfig); cfg:BDDconfig;
                   ul:(list ad); node:ad; bound:nat] : BDDconfig*ad :=
Cases bound of O => (* Error *) (initBDDconfig,BDDzero)
  | (S bound') =>
Cases (MapGet ? (negm_of_cfg cfg) node) of       (* lookup memoization map *)
  (SOME node') => (cfg,node')                     (* return the node found *)
| NONE => Cases (MapGet ? (bs_of_cfg cfg) node) of
     NONE => (* leaf node *) (if (ad_eq node BDDzero) then
         ((BDDneg_memo_put cfg BDDzero BDDone),BDDone) else
         ((BDDneg_memo_put cfg BDDone BDDzero),BDDzero))
  | (SOME (x,(l,r))) => (* internal node: recursively compute negations of *)
Cases (BDDneg_1 cfg ul l bound') of (cfgl,nodel) =>         (* child nodes *)
Cases (BDDneg_1 cfgl (cons nodel ul) r bound') of (cfgr,noder) =>
Cases (BDDmake gc cfgr x nodel noder (cons noder (cons nodel ul))) of
(cfg',node') => ((BDDneg_memo_put cfg' node node'),node')
end end end end end end.
```

Negation, as well as other algorithms, are computed by *state threading*. Recursive calls to the function, as well as calls to other functions return a new state. So the state needs to be passed around through this sequence of calls and the new state returned by each call is used in the next call.

4.7 Proofs of the Algorithms

We prove the expected semantics of `BDDmake` and the BDD algorithms in terms of Boolean functions. In addition, there are some other common things that need to be proved about all of them. Since all these functions change the store, we need to prove that the new configurations returned by them are OK. We also need to show that the new nodes returned are OK and the nodes in the old configuration that were reachable from the input list of nodes remain undisturbed in the new configuration (their semantics as Boolean functions remains unchanged). The assumptions are that the input configurations and nodes are OK.

The proofs are straightforward but long. This is specially so in the case of negation and disjunction, where we prove all these invariants together since all of them are required together for the induction to go through. As an example, we give the correctness lemma for negation, `BDDneg_1_lemma`, below. It says that for any garbage collector `gc` satisfying `gc_OK`, and for any bound `bound`, configuration `cfg`, used list `ul` and node `node`, if `bound` is greater than `(bs_var' node cfg)`, `cfg` is OK, `ul` is OK with respect to `cfg` and `node` is reachable from `ul`, then the new configuration `new_cfg` returned by `(BDDneg_1 gc cfg ul node bound)` is OK, the new node `new_node` is OK in `new_cfg`, all nodes reachable from `ul` in `cfg` are preserved in `new_cfg`, the variable at `new_node` in `new_cfg` is same as that at `node` in `cfg`, and the semantics of `new_node` in `new_cfg` is negation of the semantics of `node` in `cfg`.

```
Lemma BDDneg_1_lemma : (gc:(BDDconfig->(list ad)->BDDconfig)) (gc_OK gc)
-> (bound:nat; cfg:BDDconfig; ul:(list ad); node:ad)
(lt (nat_of_ad (var' cfg node)) bound) -> (BDDconfig_OK cfg)
-> (used_list_OK cfg ul) -> (used_node' cfg ul node) (* "node" is
                     reachable from a node in "ul" or is a leaf node *)
-> (BDDconfig_OK (Fst (BDDneg_1 gc cfg ul node bound)))
   /\ (config_node_OK (Fst (BDDneg_1 gc cfg ul node bound))
                     (Snd (BDDneg_1 gc cfg ul node bound)))
   /\ (used_nodes_preserved cfg (Fst (BDDneg_1 gc cfg ul node bound)) ul)
 (* all used nodes from old configuration are preserved in the new one *)
   /\ (ad_eq (var' (Fst (BDDneg_1 gc cfg ul node bound))
                  (Snd (BDDneg_1 gc cfg ul node bound)))
   (var' cfg node))=true (* variables at input and output nodes are same *)
   /\ (bool_fun_eq (bool_fun_of_BDD (Fst (BDDneg_1 gc cfg ul node bound))
                                    (Snd (BDDneg_1 gc cfg ul node bound)))
                  (bool_fun_neg (bool_fun_of_BDD gc cfg node))).
```

This theorem is proved in around a thousand steps. The proof of disjunction is around three times larger. We prove the above lemma using induction on `bound`. It differs from normal paper-proofs of the same algorithms in that we don't just have to reason about the BDDs on which the operations are applied, but about other parts of the store as well. We need to show that all the invariants are preserved in all the states and that other parts of the configuration remain unchanged. Also, we require the condition that the bound bound, which binds the maximum depth of recursion, is sufficiently high.

The main reason why these proofs are long is that we need to worry more about the store and less about the actual BDD on which we are computing. We require complex invariants to state that the configuration remains OK and that the store remains mostly unchanged by the operations. It might be interesting in the future to develop some general proof techniques which allow forgetting about the store and worrying only about the functional aspects of the programs.

The proofs of the BDD algorithms finally allow us to implement a tautology checker is_tauto for Boolean expressions, parameterized by the garbage collector. It works by building a BDD for the given Boolean expression, using the various BDD algorithms, and then checking that the resulting node is the leaf node BDDone. We prove the following correctness lemma is_tauto_lemma for the tautology checker, which says that is_tauto returns true exactly when the Boolean expression represents the constantly true Boolean function. Boolean expressions are defined using the inductive type bool_expr and are interpreted as Boolean functions by the function bool_fun_of_bool_expr in the obvious way. As usual, we have the assumption that the garbage collector passed as argument satisfies the predicate gc_OK.

```
Lemma is_tauto_lemma : (gc:(BDDconfig->(list ad)->BDDconfig))(gc_OK gc)
->(be:bool_expr)
(is_tauto gc be)=true<->(bool_fun_eq bool_fun_one (bool_fun_of_bool_expr be)).
```

A final note on implementing BDDs in Coq: it might seem preferable to use Typed Decision Graphs [5], a.k.a. *complement edges* [8], instead of plain BDDs. Their main value over plain BDDs is that negation operates in constant time. Here, it would allow us to dispense with the memoization table for negation, which would seem to simplify our proof work. However, disjunction is more complex to define and to reason about using complement edges. Moreover, the memoization map for disjunction, which mapped pairs of BDD nodes to BDD nodes in the plain BDD case, would map triples consisting of two BDD nodes plus a Boolean sign to pairs of BDD nodes and a Boolean sign: the latter would have to be defined under Coq as two memoization tables, one for the plus sign, one for the minus sign. This is clearly no advantage over having one memoization table for disjunction and one for negation. Still, the fact that negation is faster with complement edges means that it might be a good idea to implement and prove them, as well as any more elaborate BDD representation in the future.

5 Garbage Collection

The garbage collector implementation with which we instantiate our BDD implementation is a mark and sweep garbage collector in the style of [17]. Although reference counting is the usual choice in imperative frameworks, we believe that formalizing it in Coq is difficult, since its invariants seem more complex and a modular decomposition of its verification not entirely obvious to us. Our algorithm consists of three phases. In the mark phase, we mark all the nodes reachable from the set of nodes in the used list using depth first search. In the first sweep phase, we remove all nodes from the BDD state which are not marked and add them to the free list. In the next sweep phase, we clean the sharing and memoization maps of all invalid references (because of nodes that have now been freed).

Let us describe some Coq code from the mark phase: The following function `add_used_nodes_1` adds all the nodes reachable from a node `node` to another set of nodes `marked`. A set of nodes is implemented by the type `(Map unit)`, `unit` being the trivial type with only one element `tt`. The function `mark` computes the set of all reachable nodes by iterating over each node in the input list of used nodes.

```
Fixpoint add_used_nodes_1
  [bs:BDDstate; node:ad; marked:(Map unit); bound:nat] : (Map unit) :=
  Cases bound of O => (* Error *) (M0 unit)
    | (S bound') => Cases (MapGet ? marked node) of
         NONE => Cases (MapGet ? bs node) of NONE => marked
                    | (SOME (x,(l,r))) =>
(MapPut ? (add_used_nodes_1 bs r (add_used_nodes_1 bs l marked bound') bound') node tt)
               end
          | (SOME tt) => marked
        end end.

Definition add_used_nodes := [bs:BDDstate; node:ad; marked:(Map unit)]
  (add_used_nodes_1 bs node marked (S (nat_of_ad (bs_var' bs node)))).

Definition mark:=[bs:BDDstate;used:(list ad)](fold_right (add_used_nodes bs) (M0 unit) used).
(* here fold right iterates the function add_used_nodes for each element in used *)
```

The new BDD state is computed by restricting the domain of the original BDD state to the set of marked nodes. This can be done using the function `MapDomRestrTo` from the map library.

```
Definition new_bs := [bs:BDDstate; used:(list ad)] (MapDomRestrTo ? ? bs (mark bs used)).
```

The new free list is computed by adding to the original free list, all the nodes from the old BDD state that are not marked. The sharing and memoization maps are cleaned by looking at each entry in them and keeping only those that refer only to marked nodes. Different functions are used for cleaning different memoization maps and sharing maps, e.g., a function `clean'1` for the memoization map for negation which is of type `(Map ad)`, and a function `clean'2` (which calls the function `clean'1`) for the memoization map for disjunction (of type `(Map (Map ad))`). We omit the definitions of these functions here as they are quite long and fairly unreadable.

We prove all the results about the garbage collector mentioned in Section 4.4. We show that after garbage collection, the resulting BDD configuration is OK, by showing that the new BDD state is OK (the reducedness, ordering conditions stated in `BDDbounded` are satisfied), and that the new free list, sharing and memoization maps are OK with respect to the new BDD state. We also show that addresses that were reachable from the input list of used nodes are left undisturbed (implying that the *semantics* of all the nodes in the list as Boolean functions remains unchanged). The proofs related to the functions `clean'1`, `clean'2` mostly involve tedious arithmetical arguments related to the way addresses are stored in the maps and require proving a series of lemmas to say that the new maps contain some entry if and only if it is present in the original maps and satisfies some other conditions notably that the nodes referred to are marked. The proofs have less to do with BDDs and more to do with the details of how maps have been implemented.

6 Experimental Results

n	40	80	120	160
Coq Time	166.57	597.96	2042.38	2861.62
Caml Time	0.84	3.94	10.84	15.68
Speedup	198	152	188	182
Coq Space	33.1	68.2	39.5	48.8
Caml Space	0.4	4.1	1.0	0.9
Sp. Saving	31	12	19	30.3
#nodes	1000	10000	2000	5000

Fig. 1. Urquhart's formulae U_n

n	1	2	3	4	5	6	7	8	9	10
Coq Time	0.09	0.83	3.79	13.43	43.6	129.91	385.92	–	–	–
Caml Time	0	0.01	0.02	0.07	0.3	1.04	3.55	22.47	33.17	155.84
Speedup	–	83	190	192	145	125	109	–	–	–
Coq Space	20.7	21.6	23	27.7	34.1	49.5	84.5	–	–	–
Caml Space	36.10^{-6}	0.007	0.056	0.064	0.059	0.479	2.23	6.30	16.78	27.47
Sp. Saving	–	131	44.23	111	23	60	29	–	–	–
#nodes	8	70	270	854	2498	7006	19030	50000	129162	250000

Fig. 2. Pigeonhole formulae

We conducted several experiments to see whether this implementation is able to solve practical problems, how it scales up with problem complexity and the speed and space gained by extracting the implementation to OCaml and running the resulting, compiled programs. We give here the results for the following kinds of formulae.

- Urquhart's U-formulae [30, 28]: U_n is defined as $x_1 \Leftrightarrow (x_2 \Leftrightarrow \ldots \Leftrightarrow (x_n \Leftrightarrow (x_1 \Leftrightarrow (x_2 \Leftrightarrow \ldots \Leftrightarrow (x_{n-1} \Leftrightarrow x_n) \ldots))) \ldots)$.
- Pigeonhole formulae [28]: pig_n states that you cannot put $n+1$ pigeons in n holes with no more than one pigeon per hole.
- The 1990 IMEC benchmarks [31]: these are actual hardware verification benchmarks. We give the results for the benchmarks in the `ex` subdirectory: 3 to 8-bit multipliers (`mul03` through `mul08`), 2 to 8 bit ripple adders (`rip02, rip04, rip06, rip08`) and others (`ex2, transp, ztwaalf1, ztwaalf2`) and in the `plasco` subdirectory: `werner`. These involve checking equivalence between a naive and a more refined implementation of a circuit. These tests were also used by Harrison [19].

pb.	`mul03`	`mul04`	`mul05`	`mul06`	`mul07`	`mul08`	`rip02`	`rip04`	`rip06`	`rip08`
Coq Time	9.3	64.91	431.04	–	–	–	1.05	15.17	103.25	243.28
Caml Time	0.05	0.31	2.39	15.77	109.26	–	0	0.06	0.43	1.14
Speedup	186	209	180	–	–	–	–	253	240	213
#nodes	200	1854	8553	35852	100000	–	72	100	100	200

pb.	`ex2`	`transp`	`ztwaalf1`	`ztwaalf2`	`werner`
Coq Time	0.45	0.37	18.25	12.98	1.82
Caml Time	0	0.01	0.08	0.06	0
Speedup	–	37	228	22	–
#nodes	36	22	100	200	80

Fig. 3. IMEC benchmarks

Except for `werner` the other examples are tautologies. We measure the time and space requirement for building the BDD for a formula (the resulting node is **1** in case of tautologies). The policy we use is to allow more nodes to be allocated only if the garbage collector is unable to free any existing nodes. The behavior is not quite uniform with respect to the maximum number of nodes to be allocated because the garbage collector also requires space and so sometimes decreasing this number increases the total size requirements. We present here a few representative figures. Time is measured in seconds, and space in thousands of kilobytes. "Speedup" row gives the ratio of times takes by Coq and Caml for a formula. Space saving is computed as $(a-b)/c$ where a is the size of Coq process after building the BDD, b is the size beforehand, and c is the size of OCaml data (found by `(stat()).livewords`).

The extracted OCaml programs run around 150 times faster. Space savings are around 40-60. For pigeonhole formulae, seven pigeons is the limit for Coq and ten for the OCaml programs. Comparatively, a C version of the same algorithms is able to go up to twelve pigeons. For actual verification problems, even the Coq implementation runs in only a few seconds and difficult problems only require a few minutes (except for the multipliers, which are well known to be too hard to solve with standard BDDs). The files containing the formulas for `mul07` and `mul08` are 607 and 789 lines long respectively.

Although the Coq implementation is relatively slow and consumes a lot of memory, it is actually a pleasant surprise that such an implementation in an interpreted λ-calculus atop a simulated store runs in an acceptable amount of time and memory at all. In fact, it works on examples of quite respectable sizes, including both hard problems like pigeonhole problems and real-life problems like the IMEC benchmarks.

7 Conclusion

The import of this work is 4-fold. First, to our knowledge, this is the first complete formal proof of BDD algorithms; thanks to the Coq extraction mechanism, this yields a certified implementation in OCaml. Secondly, the BDD algorithms have not just been modeled in Coq, but completely implemented in Coq itself viewed as a programming language. This allows us and other users to replace proofs of propositional formulae in Coq's logic by mere computations in Coq's λ-calculus, and as such provides a seamless integration of BDD techniques with the Coq proof assistant. We plan to extend this to integrate a whole symbolic model checker inside Coq. Thirdly, contrary to first expectations, BDD algorithms implemented in Coq's λ-calculus atop a simulated store perform well in practice, even on hard and industrial problems. Finally, to our knowledge, this work is also the first to prove a garbage collection algorithm in full detail — what we prove is not just a model of a garbage collector, but an actual implementation that runs inside Coq itself. Furthermore, this algorithm is more complex than usual garbage collector algorithms in that it also reclaims space from memoization tables, which is not accounted for by the standard garbage collection algorithms but is needed in the context of BDDs.

Acknowledgments

We would like to thank the members of the Coq working group at INRIA Rocquencourt and at LRI, Orsay, as well as the anonymous referees for their valuable comments.

References

1. S. F. Allen, R. L. Constable, D. J. Howe, and W. E. Aitken. The semantics of reflected proof. In *LICS'90*. IEEE Computer Society Press, June 1990.
2. B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, University Paris VII, Nov. 1999. Code and Coq proofs available at `http://www.cl.cam.ac.uk/~bb236/home/coq-in-coq.tar.gz`.

3. B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq proof assistant reference manual. Version 6.2.41, available at `http://coq.inria.fr/doc/main.html`, Jan. 1999.
4. D. A. Basin and R. Constable. Metalogical frameworks. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 1–29. Cambridge University Press, 1993. Also available as Technical Report MPI-I-92-205.
5. J.-P. Billon. Perfect normal forms for discrete functions. Technical Report 87019, Bull S.A. Research Center, June 1987.
6. S. Boutin. Using reflection to build efficient and certified decision procedures. In M. Abadi and T. Ito, editors, *TACS'97*. Springer-Verlag LNCS 1281, 1997.
7. R. Boyer and J. S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*, London, 1981. Academic Press.
8. K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *DAC'90*. ACM/IEEE, 1990.
9. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C35(8), Aug. 1986.
10. O. Coudert. *SIAM : Une Boîte à Outils Pour la Preuve Formelle de Systèmes Séquentiels*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, Paris, Oct. 1991.
11. H. Goguen, R. Brooksby, and R. Burstall. Memory management: An abstract formulation of incremental tracing. In *Types'99*. Springer Verlag LNCS, 1999. Submitted.
12. G. Gonthier. Verifying the safety of a practical concurrent garbage collector. In *CAV'96*. Springer Verlag LNCS 1102, July 1996.
13. M. Gordon. Programming combinations of deduction and BDD-based symbolic calculation. Technical Report 480, University of Cambridge Computer Laboratory, Dec. 1999.
14. M. Gordon and K. F. Larsen. Combining the Hol98 proof assistant with the BuDDy BDD package. Technical Report 481, University of Cambridge Computer Laboratory, Dec. 1999.
15. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
16. M. J. C. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF, a mechanical logic of computation. Report CSR-11-77 (in 2 parts), Dept. of Computer Science, U. Edinburgh, 1977.
17. J. Goubault. Standard ML with fast sets and maps. In *ML'94*. ACM Press, June 1994.
18. J. Goubault-Larrecq. Satisfiability of inequality constraints and detection of cycles with negative weight in graphs. Part of the Coq contribs, available at `http://pauillac.inria.fr/coq/contribs/graphs.html`, 1998.
19. J. Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38, 1995.
20. G. Huet, G. Kahn, and C. Paulin-Mohring. *The Coq Proof Assistant, A Tutorial*. Coq Project, Inria, 1998. Draft, version 6.2.4. Available at `http://coq.inria.fr/doc/tutorial.html`.
21. P. Jackson. Verifying a garbage collection algorithm. In *TPHOL'98*. Springer Verlag LNCS 1479, 1998.

22. R. Kumar, K. Schneider, and T. Kropf. Structuring and automating hardware proofs in a higher-order theorem-proving environment. *Journal of Formal System Design*, 1993.
23. Z. Luo and R. Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, U. of Edinburgh, May 1992.
24. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
25. G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Functional Programming and Computer Architecture*, June 1995.
26. NIST. Common criteria for information technology security evaluation. *ISO International Standard (IS) 15408*, Jan. 2000. Version 2.1.
27. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *CADE'92*. Springer Verlag LNAI 607, June 1992.
28. F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2, 1986. Errata: Pelletier, Francis Jeffry, JAR 4, pp. 235–236 and Pelletier, Francis Jeffry and Sutcliffe, Geoff, JAR 18(1), p. 135.
29. D. M. Russinoff. A mechanically verified garbage collector. *Formal Aspects of Computing*, 6, 1994.
30. A. Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1), 1987.
31. D. Verkest and L. Claesen. Special benchmark session on tautology checking. In L. Claesen, editor, *IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI Design*, 1990.
32. R. W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artifical Intelligence*, 13(1, 2), 1980.
33. S. Yu and Z. Luo. Implementing a model checker for LEGO. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97*. Springer-Verlag LNCS 1313, Sept. 1997.