BDDs and Automated Deduction*

Jean Goubault[1] and Joachim Posegga[2]

[1] Bull Corporate Research Center, rue Jean Jaurès, 78340 Les Clayes-sous-Bois, France, (Jean.Goubault@frcl.bull.fr)
[2] Universität Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme, 76128 Karlsruhe, Germany, (posegga@ira.uka.de)

Abstract. BDDs (binary decision diagrams) are a very succesful tool for handling boolean functions, but one which has not yet attracted the attention of many automated deduction specialists. We give an overview of BDDs from an automated deduction perspective, showing what can be done with them in propositional and first-order logic, and discuss the parallels to well-known methods like tableaux and resolution.

1 Introduction

BDDs (binary decision diagrams) are a very succesful tool for handling boolean functions, which has been used extensively in hardware verification (Burch *et al.*, 1990), truth-maintenance systems (Madre & Coudert, 1991) and various other domains; often they have superseded previously-known methods. Surprisingly, they have not yet attracted the attention of many automated deduction specialists. Our aim in this paper is to show that BDDs can be profitably used in this domain, too.

Although we shall present briefly some BDD-based proof methods, the paper doesn't emphasize any technique in particular. We give an overview of BDDs from an automated deduction perspective, and demonstrate various ways in which they can be used in propositional and first-order classical logic. As regards the propositional part, we recall what can be done with BDDs in logic and how, referring the reader to the literature for complemetary results and facts. The first-order part is mostly new, and aims at showing that BDDs are a rich structure that can be used in several different ways, including tableaux-like and resolution-like techniques.

The plan of the paper is as follows: Section 2 defines what BDDs are, recalls their history briefly, and goes on defining logical operations and reduction techniques on BDDs. Section 3 discusses their theoretical and practical time and space complexities. In Section 4, we show that BDDs relate to semantic tableaux, so that both approaches can benefit from each other. We then present some principles for proving with BDDs in first-order logic, in Section 5; the first one is similar to tableaux, whereas the last one has some common points with resolution, and each one corresponds to a slight change of point of view on BDDs. Section 6 is the conclusion.

* Proc. ISMIS-94, Charlotte, N.C. Springer LNAI.

2 What are BDDs?

BDDs are nested *if-then-else*-expressions represented as graphs, which have been very successfully applied to various domains, e.g. hardware verification (see for instance (Burch *et al.*, 1990; McMillan, 1992)). These *if-then-else*-expressions are basically a case analysis over the truth value of atoms in a formula. This is a very natural way of reasoning about logical formulæ, as it resembles the way humans often deal with logic: one fixes a certain proposition and reasons about the consequences that follow if it is true, and if it is false. BDDs provide a convenient and efficient tool for implementing this form of reasoning.

The literature usually refers to an early paper by Claude Shannon (1938) as the origin of BDDs. Shannon introduced a special *if–then–else* normal form for boolean functions. The objective was to have a convenient and efficient representation for implementing boolean functions as switching circuits. The idea of using *if–then–else* expressions was not new: the principle of Shannon's cofactoring was already described in 1854 by Boole (1958). *if-then-else* normal forms were also considered by Church (1956, §24, pp. 129ff), who introduced a ternary operator of the form $[A, B, C]$ and called it a *"conditioned disjunction"*. Church's interest in this connective is based on the observation that it provides a primitive basis of propositional logic.

It is hard to command a view on the enormous amount of literature on the subject; Bryant (1992) gives a good introduction.

Various notations for expressing "*if A then B else C*" this are used in the literature; we will use here a notation that resembles Prolog's syntax:

Definition1 *if-then-else*-**connective.**

$$(A \rightarrow B; C) \stackrel{\text{def}}{=} (A \rightarrow B) \wedge (\neg A \rightarrow C)$$

(Alternatively, this can be understood as $(A \wedge B) \vee (\neg A \wedge C)$.)

In the *if-then-else*-expressions BDDs represent, no other logical connectives beside *if-then-else* are allowed. To ease exposition, we will not distinguish between a BDD and the *if-then-else*-expression it represents, but treat BDDs as ordinary logical formulæ. Recall that propositional atoms are propositional variables, and first-order atoms are applications of predicate symbols to lists of terms.

Definition2 BDDs. Let $\mathcal{L}$ be the language of (propositional/first-order) logic and $\mathcal{L}_{At}$ the atomic formulæ of $\mathcal{L}$. Assume further, that the language $\mathcal{L}$ does not contain the atomic truth values "**1**" (*true*) and "**0**" (*false*).

The set of **BDDs** is denoted by $\mathcal{BDD}$ and defined as the smallest set such that

(1) $\mathbf{1}, \mathbf{0} \in \mathcal{BDD}$
(2) *if* $\mathcal{A}, \mathcal{B} \in \mathcal{BDD}$ *and* $\phi \in \mathcal{L}_{At}$ *then* $(\phi \rightarrow \mathcal{A}; \mathcal{B}) \in \mathcal{BDD}$.

We shall use letters of the calligraphic alphabet ($\mathcal{A}, \mathcal{B}, \ldots$) to denote the elements of $\mathcal{BDD}$ in the sequel.

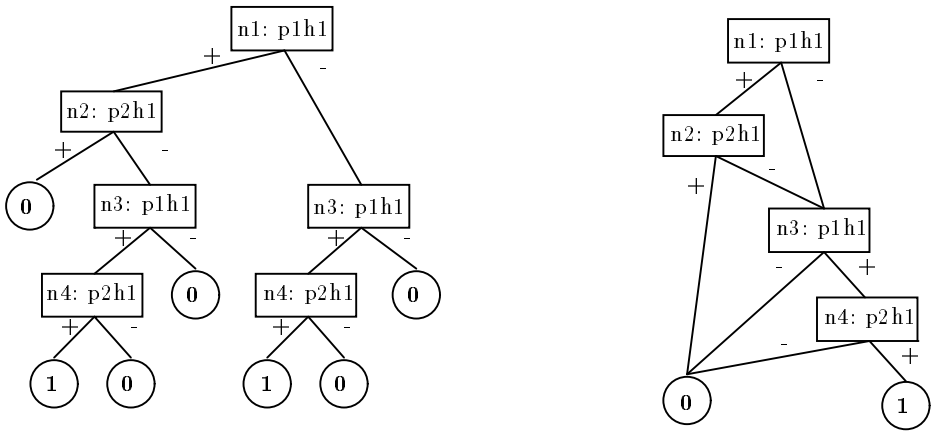

Figure 1. An *If-then-else*-expression and its representation as a BDD.

The formulæ defined above can be conveniently represented as binary trees, or binary graphs: each condition in *if-then-else*–expression is drawn as a node, and two edges, the *positive edge* (labeled with "+") and the *negative edge* (labeled with "–"), lead to the "then"-part, and the "else"-part, respectively.

Figure 1 shows an example: a binary tree that corresponds to

$$\begin{array}{l}(p1h1 \rightarrow (p2h1 \rightarrow \mathbf{0} \\ \qquad\qquad ;(p1h1 \rightarrow (p2h1 \rightarrow \mathbf{1};\mathbf{0});\mathbf{0})) \\ \quad ;(p1h1 \rightarrow (p2h1 \rightarrow \mathbf{1};\mathbf{0});\mathbf{0}))\end{array} .$$

is on the left hand side. This formula is logically equivalent to the smallest instance of the pigeon-hole formulæ $((p1h1 \rightarrow \neg p2h1) \wedge p1h1 \wedge p2h1)$. The nodes are labeled by $(n1, n2, \ldots)$ to identify them in the sequel. This formula can also be represented in a more compact way if a graph instead of a tree is used: such a graph shares identical subtrees, and is therefore much smaller. This is shown on the right hand side: "$(p1h1 \rightarrow (p2h1 \rightarrow \mathbf{1};\mathbf{0});\mathbf{0})$" occurs once in the graph, but twice in the corresponding tree. The leaves (which are, by definition, either "**1**" or "**0**") are also shared.

Note, that both the tree and the graph in Figure 1 denote the same *if-then-else*–expression; they are just different representations of it. In the sequel, we will assume BDDs are always represented as graphs that share maximally, i.e., those graphs do not contain multiple occurrences of the same subgraph. Note that the size of a graph and its corresponding tree can differ exponentially; it is in practice therefore very important to use a graph representation: graphical representation of the formulæ for pigeon-hole(7), for instance, a has about 400 nodes, but a tree 10^{34} (Posegga, 1993b).

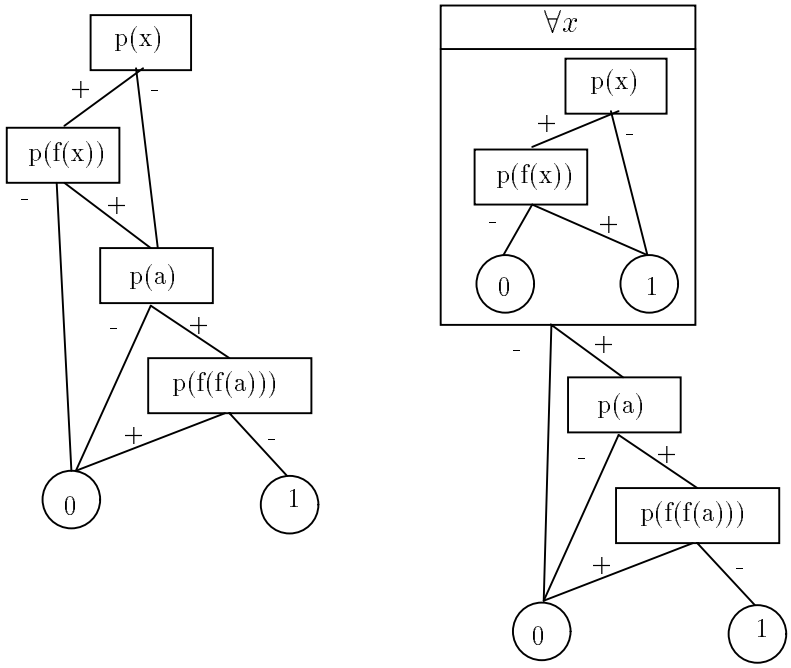

Figure 2. BDDs for $p(a) \wedge (\forall(x) p(x) \leftarrow p(f(x))) \wedge \neg p(f(f(a)))$

BDDs form a logical basis for propositional logic (see, e.g., Church (1956)). We will say that a BDD *represents a formula*, if it is logically equivalent to it. BDDs can be used to represent also quantifier-free formulæ of first-order logic (see the BDD on the left-hand side of Figure 2; the right-hand side will be discussed in Section 5.2):

Lemma 3. *For each quantifier-free formula F of first-order logic, there is a logically equivalent BDD $\mathcal{F}$.*

Paths in BDDs are an important concept that relates a BDD to its semantics; A *path* in a BDD is a sequence of signed nodes, that leads from the root to one of the leaves. Paths that lead to **1**-leaves are called **1**-paths, and paths that lead to **0**-leaves are called **0**-paths.

A path can be regarded as a conjunction of literals: the $-$-signs are treated as negation, and $+$-signs are dropped. A path is in this case either *consistent*, or *inconsistent*.

As an example, consider the right-most **1**-path of the BDD in Figure 1: "$\neg$p1h1 $\wedge$ p1h1 $\wedge$ p2h1"; this path is inconsistent.

Proposition 4. *Let $\mathcal{G} \in \mathcal{BDD}$; then*

(1) $\mathcal{G}$ is logically equivalent to the disjunction of its **1***-paths, and*
(2) $\neg\mathcal{G}$ is logically equivalent to the disjunction of its **0***-paths.*

From this proposition (see (Posegga, 1993b) for a proof) follows that a BDD "contains" both a disjunctive normal form for itself (regarded as a formula), as well as a DNF for its negation. If we regard **0**-paths as disjunctions of literals with the signs reversed instead, we can also have a CNF: the conjunction of all **0**-paths. The **1**-paths form, from this point of view, a CNF for the negated BDD.

Let us take the DNF perspective; a procedure for determining the satisfiability and falsifiability of a BDD is easily set up:

Corollary 5. *Let $\mathcal{G} \in \mathcal{BDD}$;*

(1) If there is a substitution σ, such that all **1***-paths of $\mathcal{G}\sigma$ are inconsistent, then $\mathcal{G}$ is inconsistent.*
(2) Conversely, if there is a substitution σ, such that all **0***-paths of $\mathcal{G}\sigma$ are inconsistent, then $\mathcal{G}$ is a tautology.*

Example: the BDD in Figure 1 is inconsistent, since all its **1**-paths are inconsistent.

There are several ways to convert an arbitrary logical formula into a logically equivalent BDD; one of them is to recursively rewrite the subformulæ into BDDs and combine those according to the formulæ's connectives. The following properties of the *if-then-else*–operator show how this can be done (Posegga, 1993b):

Proposition 6.

(1) $\neg(P \rightarrow \mathcal{Q};\mathcal{R}) \Leftrightarrow (P \rightarrow \neg\mathcal{Q};\neg\mathcal{R})$
(2) If $(P \rightarrow \mathcal{Q};\mathcal{R}) \in \mathcal{BDD}, F \in (\mathcal{L} \cup \mathcal{BDD})$ *and "$\circ$" is any binary logical connective, then:*
$(P \rightarrow \mathcal{Q};\mathcal{R}) \circ F \Leftrightarrow (P \rightarrow (\mathcal{Q} \circ F);(\mathcal{R} \circ F))$

Equivalence 1 shows how to move negation towards the leaves of BDDs; those negation signs can be removed at the leaves of a BDD, since those are always either **0** or **1**. Thus, a BDD can be negated by inverting its leaves, an operation that can be carried out without descending to the leaves: if an appropriate datastructure is set up, this can be implemented to require constant effort, only. 2 is a rather powerful rule that applies to every binary connective. To compute a conjunction of two BDDs $\mathcal{A}$ and $\mathcal{B}$, for example, we recursively apply rule 2 to $\mathcal{A}\wedge\mathcal{B}$ until we have moved "$\ldots\wedge\mathcal{B}$ to the leaves of $\mathcal{A}$. There it can be simplified since $\mathbf{1}\wedge\mathcal{B}$ is $\mathcal{B}$ and $\mathbf{0}\wedge\mathcal{B}$ is **0**. This means that conjunctively combining two BDDs corresponds to inserting one BDD for the **1**-leaf of the other. Symmetrically, disjunctions are handled by replacing the **0**-leaf. Both operations can be carried out in constant time.

2.1 Orderings in BDDs

BDDs, as defined above, may still contain much redundancy: the paths may have multiple occurrences of atoms, and there may be multiple occurrences of subgraphs representing propositionally equivalent subformulas.

Definition 7 Reduced BDDs.
A BDD $\mathcal{G}$ is called *reduced* if

(1) no atom occurs more than once in a path of $\mathcal{G}$, and
(2) $\mathcal{G}$ does not contain multiple occurrences of one of its subgraphs.

Example: the left graph in Figure 1 violates both conditions, the right graph violates condition 2, only.

It is of course possible to eliminate these redundancies in a given graph, but it is more reasonable to avoid them in advance while constructing a BDD. The use of *ordered* BDDs is a well-known technique to achieve this:

Definition 8 Ordered BDDs / Reduced Ordered BDDs.
Let $\mathcal{L}_{At}$ be the atoms in $\mathcal{L}$ and "$<$" be a total ordering on $\mathcal{L}_{At}$. A BDD $\mathcal{G}$ is called an *ordered BDD* if all its paths respect "$<$".

An ordered BDD which is reduced is called a *reduced, ordered BDD (ROBDD)*.

ROBDDs have the important property that all of their paths are consistent. This, together with a fixed ordering on atoms, gives a unique normal form for boolean functions. Note that, as a consequence, the derivation of a ROBDD for a given propositional formula is *NP*-hard, since every inconsistent formula will yield a ROBDD consisting of the single node "**0**". In other words: the derivation of a ROBDD for a given formula is a decision procedure for propositional logic.

The derivation of ROBDDs is described extensively in the literature (Brace *et al.*, 1990), so we only give a very brief account of it. Negation obeys the same equation as before, whereas logical connectives "$\circ$" obey new equations. Let $\mathcal{G}$ be the ROBDD $(A \rightarrow \mathcal{B}; \mathcal{C})$, and $\mathcal{G}'$ be the ROBDD $(A' \rightarrow \mathcal{B}'; \mathcal{C}')$, then:

- if $A < A'$, then $\mathcal{G} \circ \mathcal{G}' = \text{construct}(A, \mathcal{B} \circ \mathcal{G}', \mathcal{C} \circ \mathcal{G}')$;
- if $A > A'$, then $\mathcal{G} \circ \mathcal{G}' = \text{construct}(A', \mathcal{G} \circ \mathcal{B}, \mathcal{G} \circ \mathcal{C}')$;
- if $A = A'$, then $\mathcal{G} \circ \mathcal{G}' = \text{construct}(A, \mathcal{B} \circ \mathcal{B}', \mathcal{C} \circ \mathcal{C}')$,

where $\text{construct}(A, \mathcal{B}, \mathcal{C})$ is defined as $\mathcal{B}$ if $\mathcal{B}$ is identical to $\mathcal{C}$, and as $(A \rightarrow \mathcal{B}; \mathcal{C})$ otherwise. By reading these equations as rules from left to right, this recursively defines a procedure for computing $\mathcal{G} \circ \mathcal{G}'$, where the construct operation is responsible for maintaining the BDDs reduced.

By induction on the structure of quantifier-free formulæ, this leads to a procedure $\text{bdd}_< : \mathcal{L} \rightarrow \mathcal{BDD}$ that maps quantifier-free formulæ into ROBDDs[3].

[3] Algorithms for deriving ROBDDs are usually defined on propositional formulæ, only. In order to apply them to quantifier-free formulæ of first-order logic, we regard atoms simply as propositional variables.

3 Complexity

To build a non-ordered BDD is fast: we just combine BDDs for subformulæ by conjunction, disjunction or negation, which is done by constant-time operations on leaves. However, building ROBDDs, or reducing BDDs to a canonical form is more complex. We recall some results here.

First, notice that operations on ROBDDs take time that is linear (negation), or quadratic (binary operations) in the size of the operands. Better representations even yield constant-time negation. But because BDDs form a logical basis, we can solve SAT, the propositional satisfiability problem (resp. coSAT, the propositional validity problem) by building a ROBDD, then comparing it to **0** (resp. **1**). And because comparing a ROBDD with **1** or **0** is a polynomial time operation, this shows that building a ROBDD is both NP-hard and coNP-hard.

In particular, this means that ROBDDs are exponentially-sized in the worst case, and as such may need exponential time to be built. The precise bound is $O(2^n/n)$ for a ROBDD on n atomic formulæ. Unfortunately, this upper bound is also the mean value of the size of ROBDDs (Coudert, 1991), for the probability distribution that considers every ROBDD equally probable. In short, most ROBDDs are of exponential size in theory.

The good surprise is that, in practice, these huge ROBDDs very rarely occur (Coudert, 1991; McMillan, 1992). Moreover, ROBDDs have been in use in hardware verification for some time now, where the problems at hand can be NP-complete (model-checking CTL formulæ for instance), but also much harder than NP or coNP; model-checking problems in CTL* or in the modal μ-calculus (Emerson, 1990) are PSPACE-complete. Recall that PSPACE is the class of all decision problems solvable in polynomial space, and that it contains not only NP and coNP, but also the full polynomial hierarchy (Garey & Johnson, 1979). So, there must be a trick that makes ROBDDs usable in practice[4].

This trick is the choice of a good ordering for the ROBDD, and it works because in most cases, there exists a good ordering (McMillan, 1992), i.e. one that not only produces an ROBDD of reasonable size, but also that produces the intermediate ROBDDs needed to build the final one in a reasonable amount of space, too. There are some problems for which no good ordering exists (the formula describing a binary multiplier, for instance (Bryant, 1986)), and finding an optimal ordering is hard (the best known procedure needs an exponential running time (McMillan, 1992)), but usually good heuristics, related to the application domain, have been developed.

In particular, when translating formulæ written in textual form into ROBDDs, one such good ordering is based on the textual occurrences of propositional variables: if the first occurrence of variable A occurs on the left of the first occurrence of variable B, then let A be less than B. This attempts to mimick the layout of the textual formula inside the BDD and thus to control the possible explosion of the ROBDD while it is built. Put it another way, this makes a

[4] A first attempt at explaining it by average-case size estimates is given by (Dubar, 1993).

ROBDD that is close enough to the non-ordered BDD built by replacement of leaves as in the previous section.

4 BDDs and Semantic Tableaux

It is widely unknown that non-ordered BDDs are quite closely related to tableaux-based proof procedures. We shortly investigate the relation between BDDs and semantic tableaux. It will be restricted to considering propositional logic in negation normal form, but the results can be carried forward to general, first-order formulæ.

The key to considering BDDs and tableaux from a common perspective is to understand the relation between paths and branches: both have the purpose to represent potential models that are subsequently eliminated during the proof search. Assume we want to prove that a propositional formula F is inconsistent. We have already seen that the paths in a BDD are a disjunctive normal form for the BDD (seen as a logical formula), so if we have built a BDD for F, the disjunction of its paths is a disjunctive normal form for F. This is similar to semantic tableaux: the expansion of a tableau for a formula F can also be regarded as subsequently deriving a DNF for F.

Let for the rest of this section denote T_F a fully expanded tableau for a propositional formula F, and $\mathcal{F}$ denote a non-ordered BDD for F. $\#(T_F)$ denotes the set of all branches in T_F and $\Pi_{\mathcal{F}}^{\mathbf{1}}$ and $\Pi_{\mathcal{F}}^{\mathbf{0}}$ denote the set of all paths to $\mathbf{1}$–leaves and to $\mathbf{0}$–leaves in $\mathcal{F}$. The following notation will be convenient:

Definition 9. Let $k = k_1 \wedge \ldots \wedge k_n$ and $l = l_1 \wedge \ldots \wedge l_m$ be paths in a BDD or branches in a tableau, then

$$k \odot l \stackrel{\text{def}}{=} [k_1, \ldots, k_n, l_1, \ldots, l_m].$$

"$\odot$" is carried forward to sets of paths or branches K and L, by defining:

$$K \odot L \stackrel{\text{def}}{=} \{k \odot l \mid k \in K \text{ and } l \in L\}$$

Observe that $\{k\} \odot K$ means conjunctively adding k to K.

A comparison of the treatment of conjunctions and disjunctions will show that branches and paths actually correspond: $\#(T_F)$ and $\Pi_{\mathcal{F}}^{\mathbf{1}}$ are logically equivalent when seen as formulæ. This can be proven formally by structural induction; we restrict the consideration to giving the key idea for the different cases in the recursion step of such a proof (see (Posegga & Schmitt, 1994) for details).

4.1 Treatment of Conjunctions

Assume we have a formula of the form $A \wedge B$ and two fully expanded tableaux T_A and T_B; we can then build $T_{(A \wedge B)}$ by appending T_B at the end of each branch of T_A. This means: $\#\left(T_{(A \wedge B)}\right) = \#(T_A) \odot \#(T_B)$. In BDDs, a similar operation can be applied to represent a conjunction: if $\mathcal{A}$ and $\mathcal{B}$ are BDDs for

A and B, then replacing the **1**-leaf of $\mathcal{A}$ by $\mathcal{B}$ results in a conjunction of both BDDs. In terms of paths this replacement means: $\Pi^{\mathbf{1}}_{\mathcal{A}} \odot \Pi^{\mathbf{1}}_{\mathcal{B}}$ and corresponds to what happens in a tableau.

4.2 Treatment of Disjunctions

The treatment of disjunctions in BDDs is different from standard tableaux and corresponds to tableaux with *lemma generation* (see (d'Agostino, 1992) on lemma generation in tableaux). One way to achieve this is to modify the standard β–rule of a tableau calculus to

$$\frac{A \vee B}{A \mid (\neg A) \wedge B}$$

With this rule, the branches of a fully expanded tableau $T_{A \vee B}$ will be $\#(T_A) \cup (\#(T_{\neg A}) \odot \#(T_B))$. For BDDs, we can handle disjunctions by replacing **0**-leaves: a BDD for $A \vee B$ results from replacing the **0**-leaf in $\mathcal{A}$ by $\mathcal{B}$. The resulting **1**-paths are: $\Pi^{\mathbf{1}}_{\mathcal{A}} \cup (\Pi^{\mathbf{0}}_{\mathcal{A}} \odot \Pi^{\mathbf{1}}_{\mathcal{B}})$. As $\Pi^{\mathbf{0}}_{\mathcal{A}} = \Pi^{\mathbf{1}}_{\neg \mathcal{A}}$, we get the same result as in tableaux.

The above consideration shows an important difference in the representation of formulæ between BDDs and tableaux: in case of the above β–rule with lemmata, a tableau must expand the formula A and $\neg A$ separately and independently of each other. BDDs represent models and counter-models within the same structure; put simply, the replacement of **0**–leaves for representing a disjunction $A \vee B$ with BDDs attaches the countermodels of $\mathcal{A}$ to $\mathcal{B}$. Since models and counter models can be represented as efficient as models alone in tableaux, some effort during the proof search with tableaux can be avoided in BDDs.

Remark. In some cases (especially in the first-order case) it might be desirable to *avoid* lemma generation; a simple "trick" within the BDD formalism turns lemma generation off: if $\mathcal{A}$ and $\mathcal{B}$ are two BDDs, then $(L \rightarrow \mathcal{A}; \mathcal{B})$, where L is a new atom that does not appear in either (A) or (B), can also be used for representing a disjunction. Note that this treatment of disjunctions preserves satisfiability, but not equivalence. Its effect is to avoid the lemma generation effect described above. If we treat every disjunction in this way, BDDs actually simulate semantic tableaux.

5 First-order BDDs

We shall see in this section how various proof search methods in first-order logic can be rebuilt with BDDs, and how we can improve on them.

5.1 Preliminaries

To use BDDs in a first-order logic framework, we apply Herbrand's theorem. We shall stick to the tradition, and reason in terms of *unsatisfiability*. We shall assume that all formulæ of interest are in Skolem normal form. The semantic version of Herbrand's theorem is:

Proposition 10 Herbrand–Skolem–Gödel.
A formula F of the form $\forall x_1, \ldots, x_n \cdot M$, where M is quantifier-free, is unsatisfiable if and only if there exists an integer k, and k substitutions σ_1, ..., σ_k, such that $M\sigma_1 \wedge \ldots \wedge M\sigma_k$ is propositionally unsatisfiable.

If ρ_i denotes a generic renaming, mapping variables x to indexed variables x_i such that $x \neq y \vee i \neq j \leftarrow x_i \neq y_j$, we define the *$k$-fold copy* M^k of M as the formula $M\rho_1 \wedge \ldots \wedge M\rho_k$.

The previous theorem then states that F is unsatisfiable if and only if some instance of some k-fold copy of M is propositionally unsatisfiable, and we can check propositional unsatisfiability by just reducing an ordered BDD for the instance of M^k, and comparing the result to $\mathbf{0}$.

The nice thing about BDDs is that they readily contain all the structure needed to build in a wealth of various proof-search methods. We now explore some of them, considering various ways to find an instance of a BDD that reduces it to $\mathbf{0}$.

5.2 Making all 1-paths unsatisfiable

Take a BDD Φ (not necessarily ordered or reduced, thus it can be built in linear time with methods of Section 2), typically representing M^k. The only way we can choose σ such that $\Phi\sigma$ reduces to $\mathbf{0}$ (we say then that σ is a *refuting substitution*) is by having σ unify enough atomic subformulæ of Φ. This can be seen on the $\mathbf{1}$-paths: if $\Phi\sigma$ reduces to $\mathbf{0}$, then all of its $\mathbf{1}$-paths are propositionally unsatisfiable. These $\mathbf{1}$-paths can be regarded as conjunctions of literals, and are therefore unsatisfiable if and only if σ unifies two complementary (i.e, with opposite signs) literals.

This approach is reminiscent of tableau methods (Fitting, 1990), the connection method (Bibel, 1987) or the method of matings (Andrews, 1981). Indeed, if we do not consider γ and δ-rules, which govern the treatment of quantifiers (extension/amplification and Skolemization[5] in connections/matings), α and β-rules essentially build disjunctive normal forms, trying to close branches — the analogue of $\mathbf{1}$-paths — on the fly. Therefore, BDDs are actually complete expansions of tableaux with lemmata (Posegga, 1993b). The difference is that tableaux expand on demand, whereas BDDs are built entirely in memory. Because of this tight correspondence between tableaux and BDDs, BDDs provide a more compact representation of tableaux (Posegga, 1993a; Posegga & Schmitt, 1994).

Conceptually, and quite roughly, the previous method looks like this, when trying to find a propositionally unsatisfiable instance of an ROBDD Φ:

(1) initialize σ to the empty substitution []. For each $\mathbf{1}$-path in Φ:
(2) choose two complementary literals A and $-A'$ on the path.
(3) unify $A\sigma$ and $A'\sigma$, yielding σ'. If it fails, the procedure fails.
(4) otherwise, set σ to $\sigma\sigma'$, and go back to step 2.

[5] or restricted extension in the case of free-variable tableaux.

This is a non-deterministic procedure, where failure is normally implemented as backtracking (depth-first search). Of course, we don't expand paths fully and then traverse the space of all paths. Traversing the space of *partial* paths, that is, paths coming down from the root but not necessarily reaching a leaf, and extending paths when necessary, produces a much lower number of paths to close, just like in tableaux or in the connection method. A pseudo-Prolog code to achieve this is:

```
close((A → B;C),Path) :- (memberunify(¬A,Path);close(B,[A|Path])),
                         (memberunify(A,Path);close(C,[¬A|Path])).
close(LEAF,_) :- LEAF=0 ; extend(Path)
```

where the second parameter of `close/2` being the current branch in the tableau (initially `[]`) and `memberunify` denotes membership with sound unification. `extend/1` is used to build extension steps in the procedure itself. In quantifier-free BDDs, this means that we continue descending from the root of a variant of the BDD we are currently considering. This means expanding the conjunction of Proposition 10 by one step.

To narrow the search space, we can represent quantifiers in BDDs (Posegga, 1993b; Posegga & Schmitt, 1994). The idea is borrowed from γ-formulæ in semantic tableaux; we extend Definition 2 by:

(3) *if* $\mathcal{A}, \mathcal{B}, \mathcal{C} \in \mathcal{BDD}$, *then* $((\forall x \cdot \mathcal{A}) \rightarrow \mathcal{B}; \mathcal{C}) \in \mathcal{BDD}$.

This yields nested BDDs, as shown on the right-hand side of Figure 2. When we descend such nested BDDs, we do not enter the subBDDs in first instance, but use them for extensions in paths that contain non-negated occurrences of them. This reflects the γ-expansion rule of tableaux. In Prolog, this is best implemented by iterative-deepening depth-first search. Posegga also shows how this backtracking can be quite efficiently implemented by a compilation-based approach.

5.3 Eliminating 1-paths

An approach that is close to the previous one consists not in closing all **1**-paths in succession, but in *eliminating* them. To do this, we don't keep Φ static during the whole search, but reduce it with respect to the equalities between atoms induced by the found unifiers, therefore reducing the number of **1**-paths. This already induces a change of representation, as we now need to work with ROBDDs. We first define:

Definition 11 Induced equivalence relations, most general unifiers. Let $\mathcal{A}$ be a set of atoms. Any substitution σ induces an equivalence relation $\cong_\sigma$ on $\mathcal{A}$, defined by $A \cong_\sigma A'$ if and only if $A\sigma = A'\sigma$ (i.e, A unifies with A' through σ).

For every equivalence relation $\cong$ on $\mathcal{A}$, either it is induced by no substitution, or there exists a most general substitution $\sigma(\cong)$, such that all substitutions inducing $\cong$ are instances of it. We call $\sigma(\cong)$ the *most general unifier* of $\cong$.

Usually, $\mathcal{A}$ will be the set of atoms appearing in a BDD Φ. The existence and uniqueness of $\sigma(\cong)$ is a trivial generalization of Robinson's theorem on the existence and uniqueness of most general unifiers (Robinson, 1965). Notice also that, although equivalence relations and most general unifiers are close together, there is no bijection between them; however, if equivalence relations are ordered by logical pointwise implication, and substitutions are ordered by the "is an instance of" order, then $(\cong_{\_}, \sigma(\_))$ is a Galois connection.

Each time we have a new substitution σ, to see the effect it has on a BDD Φ, we can dispense with the rewriting of atoms A of Φ into $A\sigma$ followed by propositional reduction, by building the equivalence relation $\cong_\sigma$, choosing a canonical element of each equivalence class, rewriting each atom in Φ by the canonical representant of its class, and reducing the new BDD. This way, we keep applications of σ implicit. It is more interesting in practice to always rewrite every atom A in the smallest A' (the highest in the BDD) such that $A \cong_\sigma A'$. We note $A\uparrow$ this A'.

The scheme of Section 5.2 is then changed to the following, where the ROBDD Φ evolves during the course of computation:

(1) initialize σ to the empty substitution []. While there exists a **1**-path in Φ:
(2) choose two complementary literals A and $-A'$ on the path.
(3) unify $A\sigma$ and $A'\sigma$, yielding σ'. If it fails, the procedure fails.
(4) otherwise, set σ to $\sigma\sigma'$, replace Φ by the reduced version of Φ where each A has been replaced by $A\uparrow$, and go back to step 2.

Compare this with the previous scheme. Previously, we basically found a clever way of traversing the space of all **1**-paths (by factoring them through their common prefixes as partial paths) in a fixed order ("for each" in item (1)). We now have a procedure that eliminates **1**-paths in an arbitrary order: it is simply asked to choose a **1**-path and to eliminate it, until there is none remaining. This seems less smart than the previous method at first glance, because we consider full **1**-paths, instead of partial ones.

But in this scheme, the **1**-path chosen at step 1 is effectively eliminated from the new ROBDD built at step 4. Because the number of paths is exponential in the number of atoms, and each rewriting in step 4 rewrites at least one atom, then on average each pass through step 4 eliminates an exponential number of **1**-paths. Moreover, the reduction in step 4 takes time polynomial in the number of nodes of Φ, which is — this is the whole point in BDDs — usually much less than the number of **1**-paths. However, although we do more in one inference than in Section 5.2, we must pay for that by having a lower inference rate.

This approach (see (Goubault, 1993)) differs from tableaux, connections and matings in the sense that the underlying formula Φ evolves during the search for a refuting substitution, actually doing the test for propositional unsatisfiability incrementally, as new parts of the final substitution are found.

To reach completeness, as before, we have to incorporate an extension (or amplification) rule, that modifies the current BDD Φ by taking it in conjunction with a new copy $M\rho_{k+1}$ of the initial body M. Again, we can achieve this by

iterative deepening depth-first search. In general, we just need a search method that interleaves extensions and searches for instantiation fairly.

5.4 Eliminating atomic subformulæ

Looking back at the method in Section 5.3, we can see it as identifying a particular obstacle to unsatisfiability in Φ (the existence of **1**-paths), and then building on it a method aimed at eliminating the obstacle. But **1**-paths are not the only possible obstacle to unsatisfiability in ROBDDs. In particular, a propositionally unsatisfiable ROBDD not only has no **1**-paths, but sports no atomic formula either. We can therefore think of *atoms* as an alternative obstacle to propositional unsatisfiability in ROBDDs.

How do we eliminate an atom A from an ROBDD Φ? There are two cases:

- either there is a substitution σ such that $\Phi\sigma$ reduces to $\mathbf{0}$, but which unifies A with no other atom of Φ (we say that A is *unnecessary*),
- or all refuting substitutions are instances of the most general unifier of A with another atom A' in Φ.

In the second case, we can moreover restrict to consider only atoms A' that are complementary to A in Φ, in the sense that there exists a **1**-path going through $+A$ and $-A'$ or through $-A$ and $+A'$.

In the first case, assuming A is unnecessary, and $\Phi\sigma$ reduces to $\mathbf{0}$, then both $\Phi[\mathbf{1}/A]\sigma$ and $\Phi[\mathbf{0}/A]\sigma$ reduce to $\mathbf{0}$ (since the truth value of $A\sigma$ is independent of the truth values of all other substituted atoms), so, writing $\exists A \cdot \Phi$ for the reduced OBDD for $\Phi[\mathbf{1}/A] \vee \Phi[\mathbf{0}/A]$, $(\exists A \cdot \Phi)\sigma$ must reduce to $\mathbf{0}$. Conversely, if $(\exists A \cdot \Phi)\sigma$ reduces to $\mathbf{0}$, as Φ propositionally entails $\exists A \cdot \Phi$, then $\Phi\sigma$ reduces to $\mathbf{0}$.

Therefore, a proof search method striving to eliminate atoms instead of **1**-paths looks like this:

(1) initialize σ to the empty substitution $[]$. While there exists an atom A occurring in the ROBDD Φ:
(2) either replace Φ by $\exists A \cdot \Phi$ (which does not involve A any more), and go back to step 2; or choose a complementary atom A' in Φ that unifies with A with most general unifier σ'.
(3) set σ to $\sigma\sigma'$, replace Φ by the reduced version of Φ where each A has been replaced by $A\uparrow$, and go back to step 2.

At step 2, if the number of distinct most general unifiers σ' the atom A has with other atoms in Φ is p, we have the choice between $p+1$ different actions: p actions where we instantiate σ by the p different σ' and reduce Φ accordingly, plus one action where we replace Φ by $\exists A \cdot \Phi$. The procedure must do these choices non-deterministically, typically by backtracking.

Some points in this procedure are akin to resolution, or more precisely, because we have decoupled extension/amplification steps from instantiation and reduction steps, to Chang's V-resolution (Chang & Lee, 1973). Roughly, V-resolution is resolution with rigid variables, i.e, variables can only be instantiated once. The resolution rule is used both as a propositional simplification

tool, which we do not need here since ROBDDs are enough, and as a way of eliminating unnecessary atoms, by building resolvents from which these atoms have been eliminated. The **0**-paths (analogous to clauses) in $\exists A \cdot \Phi$ are just the concatenations of the **0**-paths of Φ going through $+A$ with those going through $-A$, with A excluded, as Section 4.1 has shown. Computing $\exists A \cdot \Phi$ means computing the ROBDD of all resolvents of clauses in Φ by resolving on A. All other deducible clauses are produced by the instantiation of Φ by substitutions, found by looking at unifiers of A with other atoms.

The similarity with resolution also appears in special cases of the procedure. For instance, if $p = 0$, i.e, the atom A unifies with no complementary atom in Φ, then we have but one choice to eliminate A: to transform Φ into $\exists A \cdot \Phi$. In the case where A only occurs positively (resp. negatively) on **1**-paths in Φ, this is an elimination of the pure literal $+A$ (resp. $-A$). In the general case, replacing Φ by $\exists A \cdot \Phi$ where A is not unifiable is a generalization of the pure literal rule. We call such atoms A *pure atoms*, and the transformation from Φ to $\exists A \cdot \Phi$ *pure atom elimination*.

In previous methods, we could choose whatever **1**-path we cared, to eliminate it, here we are free to choose any atom A we wish. Pure atoms should be chosen first, since they reduce the size of Φ without introducing any non-determinism. In general, A is chosen through the use of a *selection function*, just as in the linear strategy for resolution (Kowalski & Kuehner, 1971). Linear resolution and set-of-support strategies can also be adapted to the BDD case, and new techniques like instance subtraction or information control are easily implemented on BDDs (Goubault, 1994).

6 Conclusion

We pointed out how BDDs can be used in automated deduction. BDDs are a very rich structure, on which several different points of view are possible, yielding as many different ways of looking for refuting substitutions; BDDs are related to well-known proof methods, like tableaux (Section 4) or resolution (Section 5.4). We firmly believe that BDDs are not only a useful propositional tool, but a promising structuring device and implementation technique in first-order logic as well. We hope that this paper will contribute to bringing research in BDDs and automated deduction closer together. Both areas will undoubtedly benefit from that.

References

Andrews, P. (1981). Theorem Proving via General Matings. *J. ACM*, **28**(2), 193–214.

Bibel, W. (1987). *Automated Theorem Proving*. 2nd, revised edn. Vieweg.

Boole, G. (1958). *An investigation of the laws of thought, on which are founded the mathematical theories of logic and probabilities*. New York: Dover. (First Edition 1854).

Brace, K., Rudell, R., & Bryant, R. (1990). Efficient Implementation of a BDD Package. *Pages 40 – 45 of: 27th DAC.*

Bryant, R. (1986). Graph–Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, **C–35**, 677 – 691.

Bryant, R. (1992). *Symbolic Boolean manipulation with ordered binary decision diagrams.* Tech. rept. CMU.

Burch, J., Clarke, E., McMillan, K., Dill, D., & Hwang, L. (1990). Symbolic Model Checking: 10^{20} States and Beyond. *In: 5th LICS.*

Chang, C.-L., & Lee, R.-T. (1973). *Symbolic Logic and Mechanical Theorem Proving.* Computer Science Classics. Academic Press.

Church, A. (1956). *Introduction to Mathematical Logic.* Vol. 1. Princeton, NJ: Princeton University Press. 6th printing 1970 .

Coudert, O. (1991). *SIAM : Une Boîte à Outils Pour la Preuve Formelle de Systèmes Séquentiels.* Ph.D. thesis, ENST, Paris.

d'Agostino, M. (1992). Are Tableaux an Improvement on Truth–Tables? Cut–Free Proofs and Bivalence. *J. Logic, Language and Information*, **1**(3).

Dubar, P. (1993). *Estimation de tailles moyennes dans les BDD.* M.Phil. thesis, Ecole Polytechnique, Palaiseau, France.

Emerson, E. (1990). *Temporal and Modal Logic.* Elsevier. Chap. 16.

Fitting, M. (1990). *First-Order Logic and Automated Theorem Proving.* Springer.

Garey, M., & Johnson, D. S. (1979). *Computers and Intractability – A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co.

Goubault, J. (1993). Syntax Independent Connections. *In:* Basin, D. *et al.*. (ed), *Workshop on Theorem Proving with Analytic Tableaux and Related Methods.* Max Planck Institut für Informatik.

Goubault, J. (1994). Proving with BDDs and Control of Information. *In: CADE-12.*

Kowalski, R., & Kuehner, D. (1971). Linear Resolution with Selection Function. *AI*, **2**, 227–260.

Madre, J., & Coudert, O. (1991). A Logically Complete Reasoning Maintenance System Based on a Logical Constraint Solver. *In: 12th IJCAI.*

McMillan, K. (1992). *Symbolic Model Checking: an Approach to the State Explosion Problem.* Ph.D. thesis, CMU.

Posegga, J. (1993a). Compiling Proof Search in Semantic Tableaux. *Pages 67–77 of: 7th ISMIS.* LNAI, vol. 671. Trondheim, Norway: Springer.

Posegga, J. (1993b). *Deduktion mit Shannongraphen für Prädikatenlogik erster Stufe.* Infix-Verlag, FRG.

Posegga, J., & Schmitt, P. H. (1994). *Automated Deduction with Shannon Graphs.* submitted.

Robinson, J. (1965). A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, **12**(1), 23–41.

Shannon, C. (1938). A Symbolic Analysis of Relay and Switching Circuits. *AIEE Transactions*, **67**, 713 – 723.