First-order Deduction with Binary Decision Diagrams

Joachim Posegga

Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
Am Fasanengarten 5, 7500 Karlsruhe, FRG
posegga@ira.uka.de

Abstract. We show how Binary Decision Diagrams (BDDs) can be extended to full first-order logic by integrating means for representing quantifiers. A calculus based on these first-order BDDs is set up, and its soundness and completeness proofs are outlined.
The first-order BDD calculus we propose is well-suited for an efficient implementation: we propose a compilation-based approach that works by translating a first-order BDD into a Prolog program; if this program is run, it tries to show that the formula the BDD corresponds to is inconsistent.

"Contrariwise," continued Tweedledee, *"if it was so, it might be; and if it were so, it would be: but as it isn't, it ain't. That's logic."*
— Lewis Carroll

1 Introduction

Binary Decision Diagrams (BDDs) are known in Computer Science since many decades (see (Bryant, 1992) for a good introduction into BDDs); the literature usually refers to an early paper of Claude Shannon (1938) as the origin of BDDs, although this is historically not quite correct[1]. The basic idea of BDDs is to represent logical formulæ as nested *if-then-else*-expression. A logical *if-then-else*-operator forms a logical basis, if the atomic truth constants *true* and *false* are included in the language[2]. Such a normal form, represented as a graph where each nodes stands for one *if-then-else*-expression, is called a BDD. Several other criteria can be used to minimize the size of BDDs: Orderings on the nodes can be used to eliminate redundancy in the paths of a graph, which results in ordered BDDs. If additionally the size of the graph is minimized by sharing multiple occurrences of subgraphs, we get a reduced, ordered BDD (ROBDD). Such ROBDDs are a unique (canonical) normal form for boolean functions, w.r.t. a given ordering an propositional variables (Fortune *et al.*, 1978).

[1] Vlach (1993) noticed, that the principle of Shannon's co factoring was already described in 1854 by Boole (1958).
[2] Cf. Church (1956, pp. 129ff).

Over the years many fields haven taken advantage of the concept: it proved to be useful in applications like processing database queries (Hanani, 1977), pattern recognition (Bell, 1978), and diagnosis (Chang *et al.*, 1970). The so-called *branching programs* (Meinel, 1989) used in complexity theory are also closely related to BDDs. The most successful application, however, lies surely within the design of digital systems. Especially experience in hardware verification has shown that BDDs (or variants thereof) are very suitable as an underlying data-structure for proving logical properties of formulæ (Kurshan, 1990; Brace *et al.*, 1990).

Surprisingly, Automated Deduction did hardly consider BDDs in the past[3]. Quite recently, an interest in BDDs arose within Automated Deduction, as it was realized that it is possible to use BDDs not only for propositional logic, but also for full first-order deduction (Billon, 1991; Posegga & Ludäscher, 1992; Goubault, 1993). Both Billon and Goubault use ROBDDs as the underlying mechanism for deduction, whereas Posegga uses non-ordered BDDs[4]. All these approaches have the disadvantage that quantifiers of first-order formulæ cannot be represented explicitly, and that therefore some of Skolemized prenex-normal form or clausal form is required. This is problematic in that it often complicates the proof search considerably.

The present paper, which is based on the author's PhD thesis (Posegga, 1993b), shows how quantifiers can be integrated into BDDs and sets up a deduction mechanism on these first-order BDDs. The approach uses non-ordered BDDs for two reasons: Firstly, a unique normal form for first-order logic is not computable, since first-order logic in undecidable in general. Thus, the main theoretical justification for ROBDDs, namely that the provide a canonical normal form in the propositional case, is lost when considering first-order logic. Secondly, there is a practical reason for not using orderings: a first-order BDD calculus can be implemented more efficiently if free variables are allowed during the proof search, since enumerating the Herbrand universe can then be avoided. Free variables, however, are incompatible with orderings in the graphs: each application of a substitution can destroy the ordering of nodes and the minimality of a BDD; re-computing a ROBDD is, in worst case, NP-complete. Both considerations suggest that it is reasonable to consider non-ordered BDDs when dealing with first-order logic[5].

The research described in the sequel contributes both to research in BDDs and research in Automated Deduction: From the BDDs' point of view, it shows how this formalism can be lifted to a more expressive language, namely full

[3] One of the few exceptions being a paper by Ehrenfeucht and Orłowska (1967) who describe a propositional proof procedure that works on structures similar to BDDs; this method has been extended to a decidable subset of first-order logic (Orłowska, 1969).

[4] Unless explicitly stated otherwise, we always mean "non-ordered, non-reduced BDDs" when speaking of a BDD in the sequel.

[5] ROBDDs for first-order logic do make sense if free-variables during the proof search can be avoided without loosing too much efficiency (which might be the case for certain domains of application); see (Posegga & Schneider, 1993) for details.

first-order logic. From the automated deduction's point of view it offers a new approach that is similar to tableaux-based proof procedures, but more efficient than standard tableaux: it will be seen that the resulting first-order BDD calculus offers advantages in that BDDs represent models *and* counter models of a formula; this can be exploited to incorporate more efficient mechanisms for including lemmata into the proof search of tableau-based calculi. This theoretical observation has already been supported by practical results: our first-order BDD calculus has been successfully applied to hardware verification (Schneider *et al.*, 1993).

The paper is written from an automated deduction perspective and expects the reader to have a basic understanding of the working principles of free-variable tableaux (see (Fitting, 1990) for a good introduction) and BDDs (see e.g. (Bryant, 1992)). It is organized as follows: after introducing some basic notions and operations on BDDs in Section 2, a deduction mechanism is described in Section 3 and its soundness and completeness proof is outlined (3.1). After outlining a compilation-based approach to implementing the presented calculus in Section 4, we draw some conclusions from our research in Section 5.

2 Background & Basic Definitions

Let $\mathcal{L}$ be the language of first-order calculus 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*). For theoretical investigations made in this chapter we will regard BDDs simply as a logical formula, built with a special *if-then-else*-connective:

Definition 1 *if-then-else*–**connective (*ite*-connective).**
"*if-then-else*" is a new logical connective of arity three, defined as:

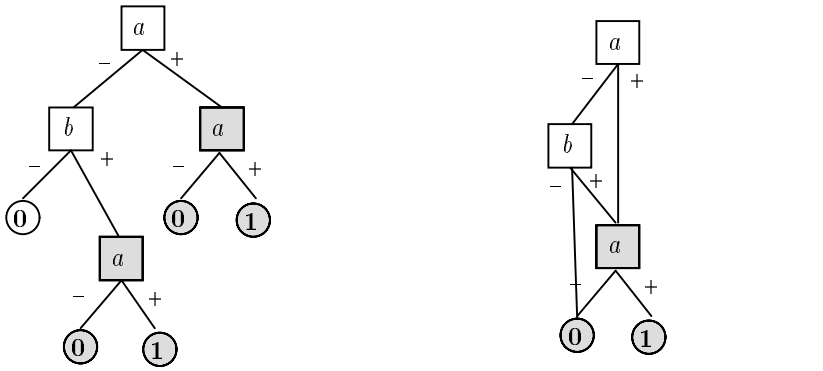
$$(A \rightarrow B; C) ::= (A \rightarrow B) \wedge (\neg A \rightarrow C)$$

One can read a formula $(A \rightarrow B; C)$ as "**if** A **then** B **else** C". (This logical operator is often defined in an *unless-then-else* form, but we will prefer the more intuitive *if-then-else*-version.)

Definition 2 First-order BDDs.
The set of *first-order BDDs* is denoted by SH and defined as the smallest set such that

(1) $\mathbf{1}, \mathbf{0} \in \mathrm{SH}$
(2) *if* $\mathcal{A}, \mathcal{B} \in SH$ *and* $\phi \in \mathcal{L}_{At}$ *then* $(\phi \rightarrow \mathcal{A}; \mathcal{B})$ *is in SH.*
(3) if $\mathcal{A}, \mathcal{B}, \mathcal{C} \in \mathrm{SH}$ then $((\forall x_1, \ldots, x_n\, \mathcal{A}) \rightarrow \mathcal{B}; \mathcal{C})$ is in SH,
where $x_1, \ldots, x_n$ are free variables occurring in $\mathcal{A}$

Thus, the first argument of the *ite*-connective in BDDs is either atomic, or a quantified BDD; the second and third arguments are always BDDs without quantifiers. It is possible to represent existential quantifiers in an analogous way



Tree representation Graph representation

Figure 1. Examples of BDDs

and their integration into the proof search described later is straightforward. However, as the framework becomes a bit simpler without existential quantifiers, we will not consider them in the sequel and assume that existential quantifiers are eliminated by Skolemization as described in a forthcoming definition (7).

It is convenient to draw BDDs (i.e. nested *ite*-expressions) as trees. An example is shown on the left hand side of Figure 1: the tree represents a BDD that is logically equivalent to $(A \vee B) \wedge A$. Each node stands for one *ite*-expression in the BDD and is labeled with its first argument; edges labeled with "+" and "–" lead to a tree representing the second and third argument, respectively. Leaves are always drawn as circles, all non-terminal nodes as squares. Nodes for representing a quantified BDDs will be called *quantified nodes* (see Figure 2 for an example).

It is useful in practise to represent BDDs as graphs by sharing common subgraphs (shown on the right hand side of the figure). This representation is much more compact and surely the better choice for an implementation. It will be seen later how such graphs can be derived. For the theoretical treatment, however, we stick to considering BDDs as logical formulæ, i.e. as trees, simply because this is handier. Note, that the difference is only a matter of representation.

For convenience, we define three projections to access the arguments of an *ite*–expression:

Definition3 Accessing Arguments of *ite*–expressions.
$[(A \to B; C)]_{\mathsf{nd}} = A$, $[(A \to B; C)]_{-} = C$, and $[(A \to B; C)]_{+} = B$
("nd" stands for "node"; B and C are called "cofactors".)

An important notion in the context of BDDs is a path; paths consist of signed nodes and can be understood as a (partial) interpretation of the atoms

occurring in a graph. We start with a formal definition basically saying that paths are sequences of nodes:

Definition 4 Paths.
A *path* π in a BDD $\mathcal{G}$ is defined as a sequence of signed subformulæ of $\mathcal{G}$. A path is denoted by an expression of the form $[\langle \mathcal{A}, v_0\rangle, \ldots, \langle \mathcal{G}_n, v_n\rangle]$, such that:

(1) $\forall i \in \{1, \ldots, n\}: \; v_i \in \{-, +\}, \mathcal{G}_i \in \mathrm{SH}$
(2) $\forall j \in \{1, \ldots, n-1\} : [\mathcal{G}_j]_{v_j} = \mathcal{G}_{j+1}$

A path is said to *start* at formula $\mathcal{A}$ (or: node $[\mathcal{A}]_{\mathsf{nd}}$), and said to end at formula $[\mathcal{G}_n]_{v_n}$ (or: node $[[\mathcal{G}_n]_{v_n}]_{\mathsf{nd}}$). $\perp(\pi)$ denotes the formula where a path π ends.
A **1**–path in a BDD $\mathcal{G}$ is a path of maximal length in $\mathcal{G}$ with $\perp(\pi) = \mathbf{1}$; thus, it starts at the root and ends at a **1**–leaf. **0**-paths are defined analogously.

As the above notation is a bit inconvenient, we will use a shorthand and write a path of the form $[\langle \mathcal{A}, v_0\rangle, \ldots, \langle \mathcal{G}_n, v_n\rangle]$ simply as $[v_0[\mathcal{A}]_{\mathsf{nd}}, \ldots, v_n[\mathcal{G}_n]_{\mathsf{nd}}]$. An example of a **1**–path in the tree or the graph of Figure 1 is $\pi = [+A, +A]$.

Definition 5 Closed Paths/Graphs.
A path π is said to be *closed* iff there is a substitution σ for the free variables in π, such that $\pi\sigma$ contains at least two atoms with complementary signs. Such a σ is called *closing substitution* for π. A path that is not closed is said to be *open*.
A BDD $\mathcal{G}$ is called *closed*, if there is a a single closing substitution σ for all **1**–paths in $\mathcal{G}$.

Paths are closely related to interpretations that map BDDs to a certain truth value. **1**–paths try to assign *true*, and **0**–paths *false*. Note, that **1**–paths play a similar role in BDDs as branches do in tableaux.
A definition for leaf-replacement in a BDD is needed in order to explain how the graph for a formula can be derived:

Definition 6 Replacement of Leaves.
Let $\mathcal{A}, \mathcal{C} \in \mathrm{SH}, \mathcal{B} \in \{\mathbf{1}, \mathbf{0}\}$; the replacement of all leaves $\mathcal{B}$ in $\mathcal{A}$ by $\mathcal{C}$ is recursively defined as follows:

$$\mathcal{A}\left[\frac{\mathcal{B}}{\mathcal{C}}\right] = \begin{cases} (A \rightarrow B\left[\frac{\mathcal{B}}{\mathcal{C}}\right]; C\left[\frac{\mathcal{B}}{\mathcal{C}}\right]) & \text{if } \mathcal{A} = (A \rightarrow B; C) \\ \mathcal{C} & \text{if } \mathcal{A} = \mathcal{B} \\ \mathcal{A} & \text{otherwise} \end{cases}$$

Note that the operation changes the second and third argument of a *ite*–term, only.

As the connective *sh*, together with the atomic truth values "**1**" and "**0**", forms a logical basis, each first-order formula can be mapped into an element of SH. The following definition describes how this can be computed:

Definition 7.

$$f2Sh(F) = \begin{cases} f2Sh(\neg A \vee \neg B) & \text{if } F = \neg(A \wedge B) \\ \langle \text{and all other standard rules for deriving negation normal form} \rangle & \\ (F \rightarrow \mathbf{1}; \mathbf{0}) & \text{if } F \in \mathcal{L}_{At} \\ (A \rightarrow \mathbf{0}; \mathbf{1}) & \text{if } F = \neg A, A \in \mathcal{L}_{At} \\ f2Sh(A)\left[\frac{\mathbf{1}}{f2Sh(B)}\right] & \text{if } F = A \wedge B \\ f2Sh(A)\left[\frac{\mathbf{0}}{f2Sh(B)}\right] & \text{if } F = A \vee B \\ ((\forall \bar{x}\, f2Sh(A)) \rightarrow \mathbf{0}; \mathbf{1}) & \text{if } F = \forall \bar{x}\, A \\ f2Sh(A\left\{\frac{x_1,\ldots,x_n}{f_1(\bar{y}),\ldots,f_n(\bar{y})}\right\}) & \text{if } F = \exists \bar{x}\, A, \text{ where } \bar{y} \text{ are the free variables in } A \text{ and } f_1, \ldots, f_n \text{ are new function symbols. (“}\{\ldots\}\text{” stands for a substitution)} \end{cases}$$

It is easily seen that the above function maps an arbitrary first-order formula without free variables into a closed BDD; due to Skolemization, the mapping preserves satisfiability, only. As an alternative to Skolemizing, it is also possible to keep the existential quantifiers and treat them analogously to universal ones.

It was mentioned earlier that BDDs are better represented as graphs than as trees; to see how such a graph can be derived, consider the above definition of *f2Sh*: some of the construction rules combine BDDs by replacing leaves of one graph by another graph, which results in a tree. If we perform this operation by replacing *edges* to leaves by *edges* to the other graph, the result is a directed, acyclic graph. In this case the computation of the above function is linear in space and time w.r.t. the length of the input formula in negation normal form. Note, that the graphs are not necessarily of minimal size as BDDs; subgraphs can still appear more than once.

From an implementation-oriented point of view, the distinction between trees and graphs is crucial, since a quite small graph can indeed represent a very huge tree. We will not want to keep an explicit representation of a tree if it is sufficient to maintain a small graph. For the propositional pigeon hole formula describing that seven pigeons do not fit into six holes, for instance, *f2Sh* yields a graph of 294 nodes; the tree representation would have 10^{42} nodes (Posegga & Ludäscher, 1992). From a theoretical point of view, on the other hand, trees are much handier than graphs: we can use a linear notation and simply regard them as logical formulæ.

2.1 BDDs and Semantic Tableaux

It is interesting to note that BDDs are quite closely related to Semantic Tableaux (Fitting, 1990): Both calculi search for proofs by construction potential models; these are ruled out as soon as contradictions in them are found. Developing a

tableau for a formula corresponds to deriving a disjunctive normal form. With BDDs, we have an *if-then-else* normal form. Both are closely related, since a DNF is contained in an *if-then-else* normal form: it can be shown that the set of all **1**-paths in a BDD for a formula corresponds to the set of all branches in a fully expanded tableau with lemmata for the same formula (see Posegga (1993b)). "tableau with lemmata" means that expanding a disjunction $A \vee B$ results in a branch containing A and another branch containing $\neg A$ and B. BDDs, however, provide a more compact representation, since counter models of a formula are "automatically" present in for of **0**-paths.

3 Deduction with First-order BDDs

The basic idea behind deduction with first-order BDDs is the following: assume we want to prove that a first-order formula F is inconsistent; then $\mathcal{F}_0 = \mathit{f2Sh}(F)$ will be inconsistent as well, since *f2Sh* preserves satisfiability. To test this, we take advantage of the following proposition:

Proposition 8. *If F is a closed first-order formula and the BDD $\mathit{f2Sh}(F)$ is closed, then F is not satisfiable.*

$\mathit{f2Sh}(F)$ is of course not *necessarily* closed if F is unsatisfiable (this is only the case for propositional formulæ). For proving first-order inconsistency, we need an iteration step that subsequently changes the graph by preserving satisfiability until we get a closed one.

The idea is borrowed from the γ–rule of free-variable tableaux: if we have an open **1**–path π in $\mathcal{F}_0$ holding a component $+\forall x\, \mathcal{G}$, then $\forall x\, \mathcal{G}$ is assumed to be true in the model that π represents. It is therefore legal[6] to conjunctively add the graph $\mathcal{G}$ (with "fresh" free variables) to this model. The operation on $\mathcal{F}_0$ to perform this is to replace the **1**–leaf $\bot(\pi)$ by $\mathcal{G}$. This is called an *extension*; if all possible extensions are exhaustively applied, it will eventually result in all **1**-paths of some graph $\mathcal{F}_j$ being closed with a substitution σ, if F is unsatisfiable.

Formally, this process can be defined by a sequence of sets of BDDs $\mathcal{F}_0, \mathcal{F}_1, \ldots$ where $\mathcal{F}_{i+1}$ is the result of performing all possible extensions on the members of $\mathcal{F}_i$:

Definition 9. Let F be an arbitrary first-order formula and $\mathcal{F} = \mathit{f2Sh}(F)$, then we recursively define a sequence of subsets of 2^{SH}:

$$\mathcal{F}_0 := \{\mathit{f2Sh}(F)\}$$

$$\mathcal{F}_{i+1} := \left\{ \mathcal{F}_i \left[\frac{\bot(\pi)}{\mathcal{G}_0\left[\frac{s}{\bar{x}'}\right]} \right] \middle| \begin{array}{l} \mathcal{G} \in \mathcal{F}_i \text{ and } \pi \text{ is a } \mathbf{1}\text{–path in } \mathcal{G} \text{ such that:} \\ (1)\ \ \pi \text{ starts at the root of } \mathcal{G}, \\ (2)\ \ \langle (\forall \bar{x}\, \mathcal{G}_0 \rightarrow \mathcal{B}; \mathcal{C}), + \rangle \text{ is an element of } \pi, \text{ and} \\ (3)\ \ \bot(\pi) = \mathbf{1}. \end{array} \right\}$$

If F is unsatisfiable, there will be an $i \in I\!N$, such that some $\mathcal{F}_i$ is closed, i.e. $\mathcal{F}_i$ is inconsistent.

[6] In the sense that is preserves satisfiability.

For an implementation, it is not reasonable to explicitly construct such sets of BDDs. The following procedure seems more reasonable instead: Starting with a formula F, we investigate all **1**–paths in the graph $f2Sh(F)$, and close them as early as possible. If there are several ways to close a branch, a choice point is generated. If a branch cannot be closed (due to not containing contradictory literals, or if all closing substitutions are not compatible with previously applied substitutions), we either backtrack to a choice point, or perform an extension by appending all **1**–paths in the quantified graph to the current path. Note that this is nothing else but an implementation-oriented view of the set-construction of Definition 9.

It is about time to consider a simple example:

Problem 10. $p(a) \quad \wedge \quad \forall x\,(p(x) \rightarrow p(f(x))) \quad \wedge \quad \neg p(f(f(a)))$

The left graph in Figure 2 shows the result of applying *f2Sh* to the above formula (the "initial graph"). The second node of this graph holds a quantified BDD for the universally quantified subformula. The only path to the **1**–leaf is not closed, so we select a universally quantified subgraph appearing with a positive prefix in this path and replacing the **1**–leaf with (an instance with fresh variables of) this graph. There is only one choice in our example.

The new graph (the "initial graph" and the "first extension" of the right hand side of Figure 2) has two paths to **1**–leaves; the leftmost path (taking the negative edge out of "p(X)") is closed with $\sigma(X) = a$, but the second remains open. After a second extension all paths are closed with $\sigma(X) = a, \sigma(Y) = f(a)$. This should give an idea how a BDD calculus works and how it can be implemented. Details on this and an efficient implementation by compiling an initial graph can be found in (Posegga, 1993b).

3.1 Soundness and Completeness of the Calculus

Definition 9 gives a complete and sound calculus for full first-order logic[7]. Soundness is easy to see, since *f2Sh* defined in 7 preserves satisfiability, and the extension operation of Definition 9, which proceeds analogously to γ-rule in free-variable semantic tableau, is also correct.

Completeness of the calculus can be proven quite elegantly with the Skolem-Herbrand-Gödel Theorem; this is outlined in the following.

Outline of the Completeness Proof The idea is to set up an analogue of the BDD-calculus on first-order formulæ in negation normal form and to show that the Skolem-Herbrand-Gödel Theorem implies the completeness of this calculus.

The $\forall$-Calculus defined in the following reflects extensions in BDDs on standard, negation normal form formulæ:

[7] If we also had existential quantifiers present, we would need an additional extension operation for elements$\langle(\exists \bar{x}\ \mathcal{G}_0 \rightarrow \mathcal{B}; \mathcal{C}), +\rangle$ in paths. Note, that it is sufficient for completeness to perform such extensions only once.

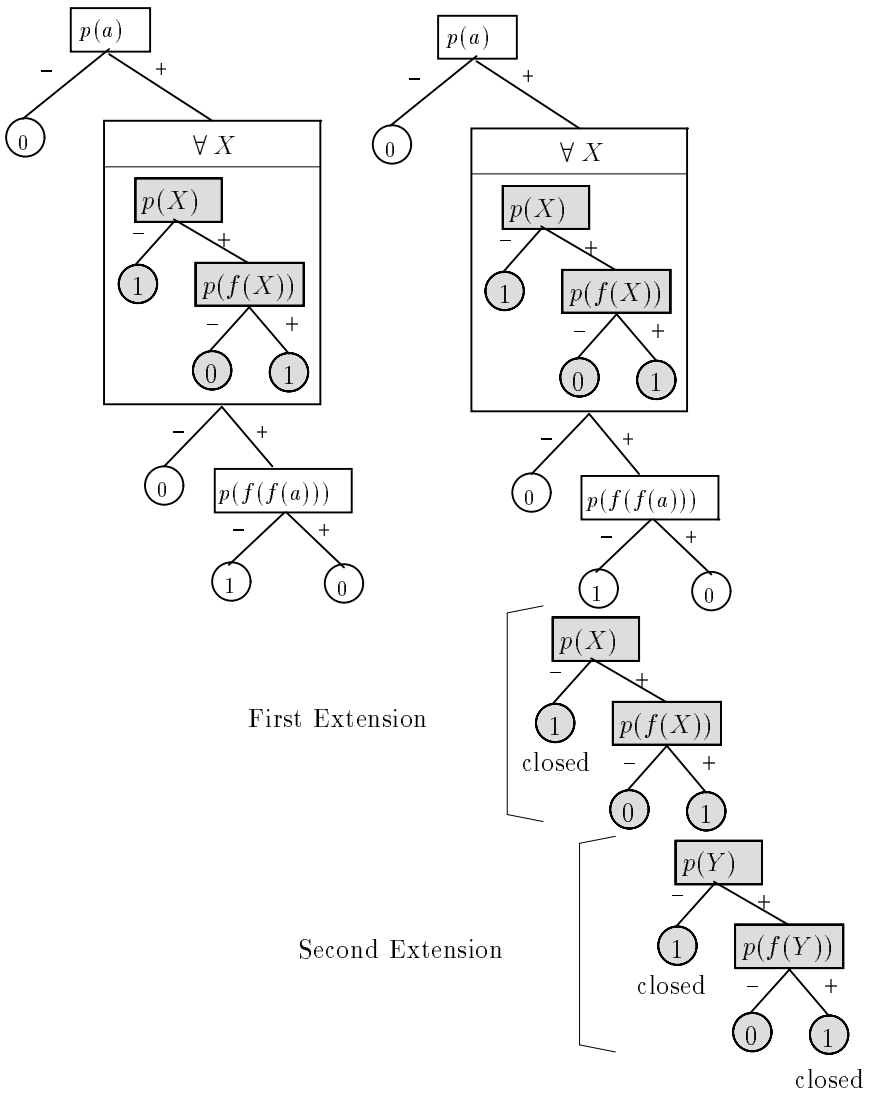
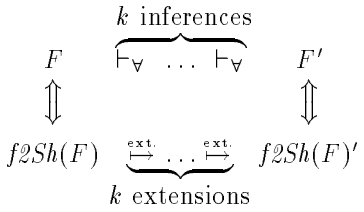


Figure 2. Proving Problem 10 is inconsistent

Definition 11 ∀-Calculus.
Let $\mathcal{L}_{\text{nnf}}$ denote the set of first-order formula in Skolemized negation normal, $F \in \mathcal{L}_{\text{nnf}}$, and assume that $rename(x, \phi)$ denotes the result of renaming the variable x in ϕ to x'. The only rule in the ∀-calculus is:

$F \vdash_\forall F[\phi \wedge rename(x, \phi')/\phi]$, if $\phi = \forall\, x(\phi')$ is a subformula of F[8].

The following diagram depicts the relation between both calculi, where the inference rules correspond:



hence, the completeness of the BDD calculus and the $\forall$-calculus are equivalent. The latter can be proven by building upon the Skolem-Herbrand-Gödel Theorem:

Proposition 12 Skolem-Herbrand-Gödel Theorem.
Assume, that $\bar{X}$ stands for a finite set of variables. If the universal closure $Cl_\forall F(\bar{X})$ for a first-order formula $F(\bar{X})$ is unsatisfiable, then there exists some $k \in \mathbb{N}$ and a grounding substitution σ, such that the conjunction

$$(F(\bar{X}_0) \wedge \ldots \wedge F(\bar{X}_k))\sigma$$

is unsatisfiable.
(σ maps all variables to ground terms, i.e., to terms from the Herbrand-universe $\mathcal{U}_F$ of F. Each $\bar{X}_i$ is a new *variable vector $X_{i,1}, \ldots, X_{i,n}$ distinct from all $\bar{X}_j$ with $j < i$.)*

Let F^{pre} denote the prenex normal form of a formula F; For proving the completeness of the $\forall$-calculus, it is sufficient to show that:

> *$\forall F \in \mathcal{L}_{\text{nnf}}$, and $k \in \mathbb{N}$: if F is inconsistent, then there exists a finite derivation $F \vdash_\forall^* F'$, such that F' can be brought into the form*
>
> $$F^{\text{pre}}(\bar{X}d_0) \wedge \ldots \wedge F^{\text{pre}}(\bar{X}_k) \wedge \Psi$$
>
> *by propositional transformations*[9]

This can be proven by structural induction.

The advantage of this procedure for proving completeness is that no infinite structures need to be considered in the proof, since the Skolem-Herbrand-Gödel Theorem already guarantees the existence of a finite derivation if a proof exists.

[8] Note, that this is nothing else than a γ-rule in a free-variable tableau calculus for NNF formulæ.

[9] The formula Ψ needs some explanation: first, it is clear that it does not cause problems for completeness, since the goal is to show an implication to the Herbrand conjunction, only. The formula Ψ basically serves to collect subformula we derive during the proof, but are not required in the conjunction we aim at. One of those are the universally quantified formulæ appearing in the input formula. To keep those is just a technical trick for remembering which formulæ can be expanded.

4 Compilation-based Implementation

First-order Deduction with BDDs can be efficiently implemented by compiling a BDD into a Prolog program that carries out the proof search at run time. This is done by translating each non-terminal node in the graph into a prolog clause. Clauses "call" each other when the corresponding nodes in the graph are connected with an edge. The program therefore constructs the paths in the graph when it runs. Consider the following problem:

Problem 13.

Axm 13.1: $p(a)$
Axm 13.2: $\neg p(f(f(a))) \wedge \neg p(g(g(a)))$
Axm 13.3: $\forall x\ (p(x) \rightarrow p(f(x))) \vee \forall (y)\ (p(y) \rightarrow p(g(y)))$

Figure 3 shows the corresponding first-order BDD. It can be closed if it is extended twice with each quantified subgraph (these are drawn separately on the right side of the picture): g2 is needed twice at the left-most **1**-leaf, and g1 twice at the right-most **1**-leaf.

A Prolog program for Problem 13 is shown in Figure 4: Each clause has three arguments: the name of the node it "implements", the path constructed so far, and a Prolog-term representing the bindings of all variables in the graph. These are known in advance, and a Prolog-term of a corresponding arity is used, where the current value of each variable is stored at a fixed position. Each clause succeeds if the graph "below" it can be closed when the current path is extended. Extensions take place if a **1**-leaf is reached and the current path cannot be closed. The introduction of new free variables in extensions is achieved by dropping the corresponding slots in the `bind`-term.

Due to lack of space we cannot discuss the compilation technique in detail; the interested reader should refer to (Posegga, 1993b). The program shown in Figure 4 runs in Quintus Prolog and finds a proof. In general, however, some form of depth-bound is needed in order to have a complete prover: Prolog's depth-first search strategy can easily cause dead loops, where an infinite number of extensions is made after choosing 'wrong' substitutions for closing paths. This, however, is easily achieved by using iterative-deepening, e.g. on the number of extensions made: backtracking is initiated if a depth-bound in `extend` is reached and alternatives for closing branches are enumerated.

The presented approach is similar to compilation-based provers like PTTP (Stickel, 1988), but more general since it does not need clausal form and "implements" quantifiers. Comparisons to PTTP have shown that proofs with compiled BDDs work roughly as fast as PTTP, but have clear advantages when non-clausal formulæ are to be proven, since the overhead involved with using CNF can be avoided. The reader is again referred to (Posegga, 1993b) for a detailed discussion including exact figures.

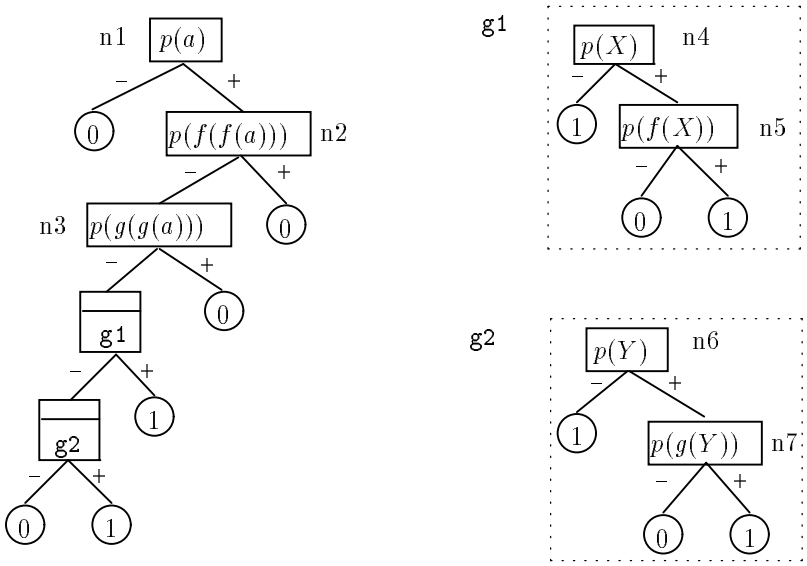


Figure 3. BDD for Problem 13

5 Conclusion

We have described a new approach to automated deduction for first-order logic based on BDDs. It was shown how propositional BDDs can be extended to represent first-order formulæ and a complete and sound calculus was presented. The main advantage of our approach over previous ones (Billon, 1991; Posegga & Ludäscher, 1992; Goubault, 1993) is that quantifier can be explicitly represented, and thus clausal form is not required.

The proposed method for lifting BDDs to first-order logic uses the same working principles as free variable semantic tableaux: the presented first-oder BDDs are closely related to fully expanded tableaux with lemmata, but provide a more efficient representation: whilst tableaux are trees, a BDD provide a much smaller graph structure. Furthermore, the presence of counter models in BDDs can be exploited for representing lemmata. Whilst the size of a tableaux can grow exponentially w.r.t. the length of a formula in negation normal form, a BDD is always linear[10].

This linearity of the representation makes it possible to apply the compilation-based implementation we described: BDDs are compiled into programs that carry out the proof search during run time. The basic idea is to generate a program that is procedurally equivalent to the test that all **1**–paths of a graph can be

[10] Note, that this does not say anything about the effort required for the proof search. It just applies to the representation the proof search is carried out upon.

```
memberunify(X,[Y|_]) :- unify(X,Y).
memberunify(X,[_|L]) :- !,memberunify(X, L).

closed_path(+Literal,Path) :- memberunify(-Literal, Path).
closed_path(-Literal,Path) :- memberunify(+Literal, Path).

extend(PATH,bind(X,Y)):- member(gamma(Node),PATH),
                         node(Node,PATH,bind(X,Y)).

%%%%%%%%%%%% initial graph %%%%%%%%%%%%
node(n1,PATH,bind(X,Y)):-
  (closed_path(+p(a),PATH);node(n2,[+p(a)|PATH],bind(X,Y))).

node(n2,PATH,bind(X,Y)):-
  (closed_path(-p(f(f(a))),PATH); node(n3,[-p(f(f(a)))|PATH],bind(X,Y))).

node(n3,PATH,bind(X,Y)):-
  (closed_path(-p(g(g(a))),PATH); node(g1,[-p(g(g(a)))|PATH],bind(X,Y))).

node(g1,PATH,bind(X,Y)):-
  node(g2,PATH,bind(X,Y)), extend([gamma(n4)|PATH],bind(X,Y)).

node(g2,PATH,bind(X,Y)):-  extend([gamma(n6)|PATH],bind(X,Y)).

%%%%%%%%%%%%%%%%%%%   Gamma graph g1 %%%%%%%%%%%%%%%%%%%
node(n4,PATH,bind(_,Y)):-
  (closed_path(-p(X),PATH) ; extend([-p(X)|PATH],bind(X,Y))),
  (closed_path(+p(X),PATH) ; node(n5,[+p(X)|PATH],bind(X,Y))).

node(n5,PATH,bind(X,Y)):-
  (closed_path(+p(f(X)),PATH) ; extend([+p(f(X))|PATH],bind(X,Y))).
%%%%%%%%%%%%%%%%%%%   Gamma graph 2 %%%%%%%%%%%%%%%%%%%
node(n6,PATH,bind(X,_)):-
  (closed_path(-p(Y),PATH) ; extend([-p(Y)|PATH],bind(X,Y))),
  (closed_path(+p(Y),PATH) ; node(n7,[+p(Y)|PATH],bind(X,Y))).

node(n7,PATH,bind(X,Y)):-
  (closed_path(+p(g(Y)),PATH) ; extend([+p(g(Y))|PATH],bind(X,Y))).
```

Figure 4. Program for Graph in Figure 3

closed; the generated program basically simulates descending down the graph and constructs paths. An implementation of the method by compiling graphs to Prolog (and also to C and Assembler language for a propositional version) has

shown good performance[11].The compilation principle does not require formulæ to be clausal form can therefore be understood as an extension of previously know approaches to compilation-based deduction.

References

Basin, D., Fronhöfer, B., Hähnle, R., Posegga, J., & Schwind, C. (eds). (1993). *Proc. 2nd Workshop on Theorem Proving with Analytic Tableaux and Related Methods.* Marseilles, France. Tech. rep. 213, Max-Planck-Institut für Informatik, Saarbrücken, Germany.

Bell, D. A. (1978). Decision Trees, tables, and lattices. *In:* Batchelor, B. G. (ed), *Pattern Recognition: Ideas in Practise.* New York: Plenum Press.

Billon, J.-P. (1991). *A New Approach of Theorem Proving for Non Clausal First Order Logic with Equality based on Generalized Shannon's Decomposition Principle.* Tech. rept. ORDA/DMA/91037. Bull Corporate Research Center, Paris, France.

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. S., Rudell, R. L., & Bryant, R. E. (1990). Efficient Implementation of a BDD Package. *Pages 40 – 45 of: Proc. 27th ACM/IEEE Design Automation Conference.* IEEE Press.

Bryant, R. Y. (1992). *Symbolic Boolean manipulation with ordered binary decision diagrams.* Tech. rept. Carnegie Mellon University. School of Computer Science.

Chang, H. Y., Manning, E., & Metze, G. (1970). *Fault Diagnosis of Digital Systems.* Vol. 62. New York: Wiley.

Church, A. (1956). *Introduction to Mathematical Logic.* Vol. 1. Princeton, New Jersey: Princeton University Press. Sixth printing 1970 .

Ehrenfeucht, A., & Orłowska, E. (1967). Mechanical Proof Procedure for Propositional Calculus. *Bull. de L'Acad. Pol. des Sci., Série des sci. math., astr. et phys.*, **XV**(1), 25–30.

Fitting, M. C. (1990). *First–Order Logic and Automated Theorem Proving.* Springer, New York.

Fortune, S., Hopcroft, J., & Schmidt, E. M. (1978). *The Complexity of Equivalence and Containment for Free Single Variable Schemes.* LNCS, vol. 62. New York: Springer–Verlag.

Goubault, J. (1993). Syntax Independent Connections. *In: (Basin* et al.*, 1993).*

Hanani, M. Z. (1977). An Optimal Evaluation of Boolean Expressions in an Online Query System. *CACM*, **20**(5), 344–347.

Kurshan, R. P. (1990). *Analysis of Discrete Event Coordination.* LNCS. New York: Springer–Verlag.

Meinel, C. (1989). *Modified Branching Programs and their Computational Power.* LNCS, vol. 370. Springer Verlag.

Orłowska, E. (1969). Automatic Theorem Proving in a Certain Class of Formulae of Predicate Calculus. *Bull. de L'Acad. Pol. des Sci., Série des sci. math., astr. et phys.*, **XVII**(3), 117 – 119.

[11] Due to the fact that BDDs can simulate semantic tableau, this compilation technique can also be applied for implementing tableau-based provers; see (Posegga, 1993a).

Posegga, J. (1993a). Compiling Proof Search in Semantic Tableaux. *Pages 67–77 of: Proc. 7th Intern. Symp. on Methodologies for Intelligent Systems.* LNAI, vol. 671. Trondheim, Norway: Springer.

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

Posegga, J., & Ludäscher, B. (1992). Towards First-order Deduction Based on Shannon Graphs. *In: Proc. GWAI.* LNAI. Bonn, Germany: Springer.

Posegga, J., & Schneider, K. (1993). Deduction with First-order BDDs. *In: (Basin* et al.*, 1993).*

Schneider, K., Kumar, R., & Kropf, T. (1993). Hardware Verification using First ORder BDDs. *In: Proc. CHDL.*

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

Stickel, M. E. (1988). A Prolog Technology Theorem Prover. CADE-9, Argonne, Ill.: Springer-Verlag.

Vlach, F. (1993). Simplification in a Satisfiability Checker for VLSI Applications. *JAR*, 115–136.