Equational Binary Decision Diagrams

Jan Friso Groote[1,2] and Jaco van de Pol[1]

[1] CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Email: JanFriso.Groote@cwi.nl, Jaco.van.de.Pol@cwi.nl
[2] Department of Computing Science, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract. We allow equations in binary decision diagrams (BDD). The resulting objects are called EQ-BDDs. A straightforward notion of reduced ordered EQ-BDDs (EQ-OBDD) is defined, and it is proved that each EQ-BDD is logically equivalent to an EQ-OBDD. Moreover, on EQ-OBDDs satisfiability and tautology checking can be done in constant time.

Several procedures to eliminate equality from BDDs have been reported in the literature. Typical for our approach is that we keep equalities, and as a consequence do not employ the finite domain property. Furthermore, our setting does not strictly require Ackermann's elimination of function symbols. This makes our setting much more amenable to combinations with other techniques in the realm of automatic theorem proving, such as term rewriting.

We introduce an algorithm, which for any propositional formula with equations finds an EQ-OBDD that is equivalent to it. The algorithm has been implemented, and applied to benchmarks known from literature. The performance of a prototype implementation is comparable to existing proposals.

1 Introduction

Binary decision diagrams (BDDs) [5, 6, 12] are widely used for checking satisfiability and tautology of boolean formulae. Applications include hardware verification and symbolic model checking. Every formula of propositional logic can be efficiently represented as a BDD. BDDs can be reduced and ordered, which in the worst case requires exponential time, but for many interesting applications it can be done in polynomial time. The reduced and ordered BDD (OBDD) is a unique representation for boolean formulae, so satisfiability, tautology and equivalence on OBDDs can be checked in constant time.

Much current research is done on extending the BDD techniques to formulae outside propositional logic. In principle, the boolean variables can be generalized to arbitrary relations. The goal now is to check satisfiability or validity of quantifier free formulae in a certain theory. The main example is the logic of *equality and uninterpreted function symbols* (EUF) [10, 7, 16]. Another example is the logic of *difference constraints* on integers or reals [13].

EUF formulae have been successfully applied to the verification of pipelined microprocessors [8, 7] and of compiler optimizations [16]. In these applications, functions can be viewed as black boxes that are connected in different ways. Hence the concrete functions can be abstracted from, by replacing them by uninterpreted function symbols (i.e., universally quantified function variables). It is clear that if the abstracted formula is valid, then the original formula is. However, the converse is not true, e.g. $x + y = y + x$ is valid, but its abstract version $F(x, y) = F(y, x)$ is not.

Two methods for solving EUF formulae exist. The first method is based on two observations by Ackermann [1]. First, the function variables can be eliminated, essentially by replacing any two subterms of the form $F(x)$ and $F(y)$ by new variables f_1 and f_2, and adding functionality constraints of the form $x = y \rightarrow f_1 = f_2$. The second observation is the *finite domain property*, which states that the resulting formula is satisfiable if, and only if, it is satisfiable over a finite domain. Given an upper bound n on this domain, each domain variable can be encoded as a vector of $\lceil \log(n) \rceil$ bits. In this way the original problem is reduced to propositional logic, and can be solved using existing BDD techniques.

The second method extends the BDD data structure, by allowing equations in the nodes of a BDD, instead of boolean variables only. By viewing all atoms as distinct variables, the BDD algorithms can still be used to construct a reduced ordered BDD. Contrary to the propositional case, a path in these OBDDs can be inconsistent, for instance because it violates transitivity constraints. As a consequence, all paths of the resulting OBDD have to be checked in order to conclude satisfiability.

Ultimately, we are interested in the symbolic verification of distributed systems, using high-level descriptions. This involves reasoning about data types (specified algebraically) and control (described by boolean conditions on data). Properties of the system are described using large boolean expressions. We want to use BDD-techniques in order to prove, or at least simplify, boolean expressions containing arbitrary relation and function symbols. In this setting, abstraction doesn't work, as it doesn't preserve logical equivalence. Without abstraction, Ackermann's function elimination cannot be applied, and the finite domain property doesn't hold.

We therefore turn to the second method, allowing equations in the BDD nodes. We will give a new definition of "ordered", such that in ordered BDDs all paths will be consistent. The advantage is that on ordered BDDs with equations, the satisfiability check can be done in constant time. The contribution of this paper is an intermediate step towards the situation where arbitrary relations and function symbols in BDDs are allowed. We restrict to the case of equations, without function symbols.

Technical Contribution. In Section 2 we introduce EQ-BDDs, which are BDDs whose internal nodes may contain equations between variables. We extend the notion of orderedness so that it covers the equality laws for reflexivity, symmetry, transitivity and substitution. The main idea is that in a (reduced) ordered EQ-

BDD (EQ-OBDD) of the form ITE($x = y, P, Q$), y may not occur in P; this can be achieved by substituting occurrences of y by x. By means of term rewriting techniques, we show that every EQ-BDD is equivalent to an EQ-OBDD.

Contrary to OBDDs, EQ-OBDDs are not unique, in the sense that different EQ-OBDDs may still be logically equivalent, so equivalence checking on EQ-OBDDs cannot be done in constant time. However, we show that in an EQ-OBDD, each path from the root to a leaf is consistent. As a corollary, **0** is the only contradictory EQ-OBDD, and **1** is the only tautological one. Every other EQ-OBDD is satisfiable. So satisfiability and tautology checking on EQ-OBDDs can still be done in constant time.

We present an algorithm for converting propositional formulae with equality into an EQ-OBDD in Section 3. Usually a bottom-up algorithm is used, based on Bryant's APPLY algorithm [5], which implements the logical connectives on OBDDs in polynomial time. In the presence of equalities, APPLY would involve new substitutions, which possibly cause a reordering of the subformulae.

Instead, we use a generalization of the top-down method (cf. [12]). The inefficiency usually attributed to this top-down approach is avoided by using memoization techniques and maximal sharing. We have made a prototype implementation in C, which uses the `ATerm` library [4] to manipulate terms in maximally shared representation. We applied this implementation on the benchmarks used in [16, 19]. It appears that our ideas yield a feasible procedure, and that the performance is comparable to the approach in [16].

In EQ-BDDs, interpreted function symbols can be incorporated straightforwardly. A complete term rewrite system for the algebraic data part can be used to reduce the nodes. This always leads to equivalent formulae, but completeness of the method is lost. In future work we plan to investigate under which circumstances completeness can be regained. The fact that equality is incorporated directly, instead of encoded, can give BDD-techniques a much more prominent place in interactive theorem provers like PVS [15]. The fact that the performance of our prototype implementation is comparable with existing proposals indicates that extendibility does not necessarily come with a loss in efficiency.

Related work. After Ackermann [1] proved decidability of quantifier free logic with equality, Shostak [18] and Nelson and Oppen [14] provided practical algorithms for the validity check, based on the congruence closure. Those authors used a transformation to disjunctive normal forms. In [8] this transformation is avoided, by dealing more efficiently with boolean combinations; in particular they incorporate case splitting as in the Davis-Putnam procedure. We next consider papers based on BDDs, that either use the aforementioned method based on the finite domain property, or allow arbitrary atoms in the BDD nodes.

Two recent papers [7, 16] refine the method based on finite domains. The main contribution of Bryant et al. [7] is to distinguish between function symbols that occur in *positive equations* only (p-symbols) and other function symbols (g-symbols). This allows to restrict attention to maximally diverse interpretations, in which p-symbols can be interpreted by a fixed value. Also Ackermann's function elimination is improved. Pnueli et al. [16] provide heuristics to obtain lower

estimates for the domains. These estimates are also obtained by distinguishing between positive and negative occurrences of equations. Both methods rely on the finite domain property, whereas our solution avoids this.

The other method is closer to our approach. Goel et al. [10] avoid bit vectors for finite domains, by introducing boolean variables e_{ij}, representing the equation $x_i = x_j$. So their method doesn't rely on the finite model property. Similarly, Møller et al. [13] allow difference constraints of the form $x - y \leq c$ in the BDD nodes, with c an integer or real constant. In case the underlying domain consists of integers or reals, $x = y$ can be encoded as $x - y \leq 0 \wedge y - x \leq 0$, leading to two different nodes. For other underlying domains, such as natural numbers or lists, this encoding is not possible, where our approach works for equality in any domain.

Both [10] and [13] first reduce a formula to OBDD, viewing all boolean terms as different variables. Although the nodes on a path are all different after this operation, a path can still be inconsistent, for instance by violating transitivity. Parts of the OBDD are inaccessible, so in general the OBDD is too large. The OBDD can be further reduced in order to check satisfiability (this is called path-reduced in [13]), but this involves the inspection of all paths, of which there can be exponentially many. Indeed, in [10] it is proved that deciding whether an OBDD with e_{ij}-variables has a satisfaction that complies with transitivity is NP-complete. In our case, the paths in the resulting EQ-OBDD are consistent and the test for satisfiability on EQ-OBDDs requires constant time only.

Another approach, mentioned in the full version of [7], considers the addition of transitivity constraints to a formula. Adding all of them usually leads to a blow-up of the BDD. A heuristics is presented to prune the set of needed transitivity constraints. In our approach transitivity constraints are generated on the fly when needed, by performing proper substitutions.

In the implementation, the fundamental data structure is a maximally shared term, partly consisting of boolean connectives, and partly of BDD-nodes. This resembles the Binary Expression Diagrams (BEDs) of [2], for the pure boolean case. We have not thoroughly studied the relationship between our top-down algorithm and their *up-one*. In [17] it is indicated how such a comparison could be made in principle, by using term rewriting theory on strategies. Also a thorough comparison with the algorithm in [8] would be interesting.

2 EQ-BDDs

We now define a syntax for formulae. First assume disjoint sets P and V. Members of P are called proposition (boolean) variables (typically p, q, ...) and V contains domain variables (typically x, y, z, ...).

Definition 1. *Formulae are expressions satisfying the following syntax:*

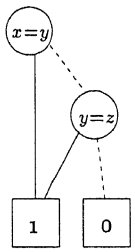
$$\Phi \ ::= \ \mathbf{0} \mid \mathbf{1} \mid P \mid V = V \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathrm{ITE}(\Phi, \Phi, \Phi)$$

We use $x \neq y$ as an abbreviation of $\neg(x = y)$. In order to avoid confusion, we write $\equiv$ for syntactic equality, so $x \equiv y$ means that x and y are the same variable.

An interpretation consists of a non-empty domain D and interpretation functions $I : V \to D$ and $J : P \to \{\mathbf{0}, \mathbf{1}\}$. Then the semantics of Φ, denoted by $\Phi_I^J \in \{\mathbf{0}, \mathbf{1}\}$, can be defined straightforwardly. In particular, $\mathrm{ITE}(x, y, z)_I^J = y_I^J$ if $x_I^J = \mathbf{1}$, otherwise it equals z_I^J. Equality is interpreted as the identity relation by defining $(x = y)_I^J$ as $\mathbf{1}$ if $I(x) = I(y)$, $\mathbf{0}$ otherwise. Now D, I, J forms a model for Φ iff $\Phi_I^J = \mathbf{1}$. Φ is satisfiable iff it has a model and it is tautological (or: universally valid) iff all interpretations are models. Φ and Ψ are logically equivalent iff they have the same models. A theory is a set of formulae. Given a theory S, we write $S \models \Phi$ iff all models for S are models of Φ. We rely on the following lemma, which is a theorem of Shostak [18], specialized to the case without function symbols.

Lemma 2. *Let S be a set of equalities and T a set of inequalities. Then $S \cup T$ is satisfiable if and only if for all $x \neq y \in T$, $x = y$ is not in the reflexive, symmetric and transitive closure of S.*

We now turn to the study of EQ-BDDs, which can be seen as a subset of formulae, and consider arbitrary formulae in Section 3. A binary decision diagram (BDD [6, 12]) is a DAG, whose internal nodes contain guards, and whose leaves are labeled $\mathbf{0}$ (low, false) or $\mathbf{1}$ (high, true). Each node contains two distinguished outgoing edges, called low and high. In ordinary BDDs, the guards solely consist of proposition variables. The only difference between ordinary BDDs and EQ-BDDs is that in the latter, a guard can also consist of equations between domain variables. EQ-BDDs can be depicted as follows (the low/false edges are dashed):



We reason mainly about EQ-BDDs as a restricted subset of formulae, although in implementations we always treat these formulae as maximally shared DAGs. There are constants to represent the nodes $\mathbf{0}$ or $\mathbf{1}$. Furthermore, we use the if-then-else function $\mathrm{ITE}(g, t_1, t_2)$ where g is a guard, or label of a node in the BDD, t_1 is the high node and t_2 is the low node. Guards can be proposition variables in P, or equations of the form $x = y$ where x and y are domain variables (V).

Definition 3. *We define the set G of guards and B of EQ-BDDs,*

$$G ::= P \mid V = V$$
$$B ::= \mathbf{0} \mid \mathbf{1} \mid \mathrm{ITE}(G, B, B)$$

The EQ-BDD depicted above can be written as: $\mathrm{ITE}(x = y, \mathbf{1}, \mathrm{ITE}(y = z, \mathbf{1}, \mathbf{0}))$.

In order to compute whether an EQ-BDD is tautological or satisfiable, it will first be ordered. In an *ordered* EQ-BDD, the guards on a path may only appear in a fixed order. To this end, we impose a total order on $P \cup V$ (e.g. $x \succ p \succ y \succ z \succ q$). This order is extended lexicographically to guards as follows:

Definition 4 (Order on guards).

$$\begin{aligned} p \succ q &\textit{ as given above} \\ (x = y) \succ p &\textit{ if, and only if, } x \succ p \\ p \succ (x = y) &\textit{ if, and only if, } p \succ x \\ (x = y) \succ (u = v) &\textit{ if, and only if, either } x \succ u, \textit{ or } x \equiv u \textit{ and } y \succ v. \end{aligned}$$

Given this order, we can now define what we mean by an ordered EQ-BDD. We use some elementary terminology from term rewrite systems (TRSs), which can for instance be found in [11, 3]. In particular, a *normal form* is a term to which no rule can be applied. A system is *terminating* if no infinite rewrite sequence exists.

Definition 5. *An EQ-BDD is ordered if, and only if, it is a normal form w.r.t. the following term rewrite system, called* ORDER. *An EQ-OBDD is an ordered EQ-BDD.:*

1. $\mathrm{ITE}(G, T, T) \to T$.
2. $\mathrm{ITE}(G, \mathrm{ITE}(G, T_1, T_2), T_3) \to \mathrm{ITE}(G, T_1, T_3)$.
3. $\mathrm{ITE}(G, T_1, \mathrm{ITE}(G, T_2, T_3)) \to \mathrm{ITE}(G, T_1, T_3)$.
4. $\mathrm{ITE}(G_1, \mathrm{ITE}(G_2, T_1, T_2), T_3) \to \mathrm{ITE}(G_2, \mathrm{ITE}(G_1, T_1, T_3), \mathrm{ITE}(G_1, T_2, T_3))$, *provided* $G_1 \succ G_2$.
5. $\mathrm{ITE}(G_1, T_1, \mathrm{ITE}(G_2, T_2, T_3)) \to \mathrm{ITE}(G_2, \mathrm{ITE}(G_1, T_1, T_2), \mathrm{ITE}(G_1, T_1, T_3))$, *provided* $G_1 \succ G_2$
6. $\mathrm{ITE}(x = x, T_1, T_2) \to T_1$.
7. $\mathrm{ITE}(y = x, T_1, T_2) \to \mathrm{ITE}(x = y, T_1, T_2)$, *provided* $x \prec y$
8. $\mathrm{ITE}(x = y, T_1[y], T_2) \to \mathrm{ITE}(x = y, T_1[x], T_2)$, *if* $x \prec y$ *and* y *occurs in* T_1.

Rules 6–8 capture the properties of equality, viz. reflexivity, symmetry, and substitutivity. From these rules, transitivity can be derived, as we demonstrate in Figure 1 (we assume $x \prec y \prec z$). Note that in rule 8 *all* instances of y in T_1 are replaced by x. From a term rewriting perspective this is non-standard, because it is a non-local rule.

In a normal form no rewrite rules are applicable. Hence it is easy to see that in an ordered EQ-BDD, the guards along a path occur in strictly increasing order (otherwise rule 2/3/4/5 would be applicable) and in all guards of the form $x = y$, it must be the case that $x \prec y$ (otherwise rule 6/7 would be applicable). Note that the transformations indicated by the rules are sound, in the sense that they yield logically equivalent EQ-BDDs.

We prove that each EQ-BDD is equivalent to an EQ-OBDD, by showing that the TRS ORDER always terminates. The termination proof uses the powerful *recursive path ordering* (RPO) [9]. For RPO comparisons, we view $\mathrm{ITE}(g, t_1, t_2)$

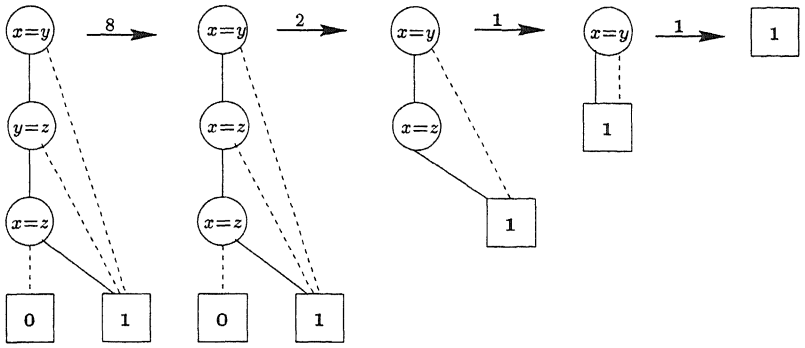


Fig. 1. Derivation of transitivity of equality in EQ-BDDs

as $g(t_1, t_2)$. RPO needs an ordering on the function symbols. For this we just use the total order on guards of Definition 4, extended with $\mathbf{1}, \mathbf{0} \prec g$ for all guards. In this case, RPO specializes to the following relation:

Definition 6. $s \equiv f(s_1, s_2) \succ_{\text{rpo}} t$ *iff* $t \equiv \mathbf{0}$ *or* $t \equiv \mathbf{1}$, *or* $t \equiv g(t_1, t_2)$ *and one of the following holds:*

- *(I)* $s_1 \succeq_{\text{rpo}} t$, *or* $s_2 \succeq_{\text{rpo}} t$;
- *(II)* $f \succ g$ *and* $s \succ_{\text{rpo}} t_1$ *and* $s \succ_{\text{rpo}} t_2$;
- *(III)* $f \equiv g$ *and either* $s_1 \succ_{\text{rpo}} t_1$ *and* $s_2 \succeq_{\text{rpo}} t_2$, *or* $s_2 \succ_{\text{rpo}} t_2$ *and* $s_1 \succeq_{\text{rpo}} t_1$.

Here $x \succeq_{\text{rpo}} y$ means: $x \succ_{\text{rpo}} y$ or $x \equiv y$. Usually in clause (III) the multiset or lexicographic extension is used, but this is not needed for our purposes. From the literature, it is well known that $\succ_{\text{rpo}}$ is an order (in particular the relation is transitive), which is well-founded (because $\succ$ on guards is) and monotone, so it is useful in proving termination.

Lemma 7. *The rewrite system* ORDER *is terminating.*

Proof. It is straightforward to show that rule 1–8 are contained in $\succ_{\text{rpo}}$ (for rule 8 monotonicity of $\succ_{\text{rpo}}$ is used). From this termination follows. □

Theorem 8. *Every EQ-BDD is equivalent to some EQ-OBDD.*

Traditional OBDDs are unique representations of boolean functions, which makes them useful for checking equivalence between formulae. For EQ-OBDDs, however, this uniqueness property fails, as the following example shows.

Example 9. Let $x \prec y \prec z$. Consider the EQ-BDDs $\text{ITE}(x = y, \mathbf{1}, \text{ITE}(y = z, \mathbf{0}, \mathbf{1}))$ and $\text{ITE}(x = z, \mathbf{1}, \text{ITE}(y = z, \mathbf{0}, \mathbf{1}))$. These represent the predicates $y = z \to x = y$ and $y = z \to x = z$, which are logically equivalent. Both are ordered, because no rewrite rule is applicable. But they are not identical. □

Although EQ-OBDDs do not have the uniqueness property, satisfiability or tautology checking can still be done in constant time. The rest of this section is devoted to the proof of this statement.

Definition 10. Paths *are sequences of* 0*'s and* 1*'s. We let letters* α, β *and* γ *range over paths, and write* ε *for the empty sequence,* $\alpha.\beta$ *for the concatenation, and* $\alpha \sqsubseteq \beta$ *if* α *is a prefix of* β. *With* $seq(T)$ *we denote the sequences that correspond to a path in EQ-BDD* T. *For a path* $\alpha \in seq(T)$ *we write* $T|_\alpha$ *for the* guard at the end of path α, *inductively defined by:*

- $\mathrm{ITE}(G,T,U)|_\varepsilon = G$.
- $\mathrm{ITE}(G,T,U)|_{1.\alpha} = T|_\alpha$ *(the high branch).*
- $\mathrm{ITE}(G,T,U)|_{0.\alpha} = U|_\alpha$ *(the low branch).*

We also define the theory up to the node corresponding to path $\alpha \in seq(T)$, *notation* $Th(T,\alpha)$, *inductively on an EQ-BDD* T*:*

- $Th(T,\varepsilon) = \emptyset$.
- $Th(T,\alpha.1) = Th(T,\alpha) \cup \{T|_\alpha\}$.
- $Th(T,\alpha.0) = Th(T,\alpha) \cup \{\neg T|_\alpha\}$.

Finally, $\alpha \in seq(T)$ *is called* consistent *iff* $Th(T,\alpha)$ *is satisfiable.*

Example 11. Let $T \equiv \mathrm{ITE}(x = y, \mathbf{1}, \mathrm{ITE}(y = z, \mathrm{ITE}(x = z, \mathbf{1}, \mathbf{0}), \mathbf{1}))$. Then the guard at path 0.1 is: $T|_{0.1} \equiv x = z$. The theory at that point is: $Th(T, 0.1) = \{x \neq y, y = z\}$ which is satisfiable, so 0.1 is consistent. □

The analysis of EQ-OBDDs depends on the following rather syntactic lemma. The first states that in EQ-OBDDs y does not occur below the high branch of $x = y$; the second states that y does not occur positively above $x = y$.

Lemma 12. *Let* T *be an EQ-OBDD, and* $\alpha, \beta \in seq(T)$ *be consistent paths.*

1. *If* $T|_\alpha \equiv x = y$ *and* $\alpha.1 \sqsubseteq \beta$, *then* $T|_\beta \not\equiv z = y$ *and* $T|_\beta \not\equiv y = z$.
2. *If* $T|_\alpha \equiv x = y$ *and* $\beta.1 \sqsubseteq \alpha$, *then* $T|_\beta \not\equiv z = y$ *and* $T|_\beta \not\equiv y = z$.
3. *If* $Th(T,\alpha) \models x = z$ *and* $x \prec z$, *then for some* y, $y = z \in Th(T,\alpha)$.

Proof. (1) If $T|_\beta$ contains y, rewrite step 8 would be applicable, which contradicts orderedness.

(2) If $T|_\beta \equiv z = y$ rewrite step (8) is applicable, contradicting orderedness. Assume $T|_\beta \equiv y = z$. Note that $x \prec y$, as $x = y$ appears in the EQ-OBDD, so $x = y \prec y = z$. Hence, on the path between the nodes labeled with $y = z$ and $x = y$, at least one of the steps (4,5) would be applicable. This contradicts orderedness of T.

(3) Let $Th(T,\alpha) \models x = z$. Note that $Th(T,\alpha)$ is satisfiable, but $Th(T,\alpha) \cup \{x \neq z\}$ is not. Hence by two applications of Lemma 2, $x = z$ is in the reflexive, symmetric, transitive closure of the positive equations in $Th(T,\alpha)$. I.e. there exist n and x_i $(0 \leq i \leq n)$, such that $x_0 \equiv x$, $x_n \equiv z$ and for all i $(0 \leq i < n)$, $x_i = x_{i+1} \in Th(T,\alpha)$ or $x_{i+1} = x_i \in Th(T,\alpha)$. Because $x \prec z$, we have $n \geq 1$.

Consider the last equation in this sequence, which is either $x_{n-1} = z \in Th(T, \alpha)$, in which case we are done, or it is $z = x_{n-1} \in Th(T, \alpha)$. In this case, x_{n-1} doesn't occur in any other equation (it cannot occur positively above $z = x_{n-1}$ in T by (2), nor can it occur below it by (1)). Hence $n = 1$ and $z = x \in Th(T, \alpha)$. This contradicts orderedness of T, because $x \prec z$. □

We can now prove that each guard in an EQ-OBDD is logically independent from those occurring above it.

Lemma 13. *Let T be an EQ-OBDD and let $\alpha \in seq(T)$ be consistent. Then*

1. $Th(T, \alpha) \not\models T|_\alpha$ *and*
2. $Th(T, \alpha) \not\models \neg T|_\alpha$.

Proof. If $T|_\alpha \equiv p$ ($p \in P$), then by orderedness, p does not occur in $Th(T, \alpha)$, so the lemma follows (this is similar to the traditional BDD-case). Now let $T|_\alpha \equiv x = z$. Hence, $x \prec z$.

(1) Assume $Th(T, \alpha) \models x = z$. By Lemma 12.3, for some y, $y = z \in Th(T, \alpha)$. Then rewrite step 8 is applicable, which contradicts orderedness.

(2) Assume $Th(T, \alpha) \models x \neq z$. Using Lemma 2 it can be proved that for some y and v, $Th(T, \alpha) \models \{x = y, v = z\}$ and either $y \neq v \in Th(T, \alpha)$ or $v \neq y \in Th(T, \alpha)$. By Lemma 12.2, no positive equations containing z occur in $Th(T, \alpha)$, so $z \equiv v$. Now if $z \neq y \in Th(T, \alpha)$, $z = y$ occurs above $x = z$ in the ordered EQ-BDD T, so $z \prec x$, contradicting $x \prec z$. Hence, $y \neq z \in Th(T, \alpha)$. Note that as T is ordered and $y = z$ occurs above $x = z$, $y \prec x$. Now by Lemma 12.3, for some w, $w = x \in Th(T, \alpha)$. But then rewrite step 8 would be applicable, which contradicts orderedness. □

Theorem 14. *Satisfiability and tautology on EQ-OBDDs can be checked in constant time.*

Proof. Using Lemma 13 it can be proved that each path to a leaf in an EQ-OBDD is consistent, so all leaves are reachable by some interpretation. Hence if the EQ-OBDD is a tautology, all leaves must be syntactically equal to **1**, and by rule (1) of ORDER, the EQ-OBDD must be the node **1**. In a similar way, the only contradictory EQ-OBDD is **0**. Hence an EQ-OBDD is satisfiable if, and only if, it is syntactically different from **0**. □

3 Algorithm for Checking Tautology and Satisfiability

We are now interested in constructing EQ-BDDs out of formulae. In traditional BDDs, a formula is transformed into an OBDD in a bottom-up fashion. Given two ordered BDDs, the logical operations (conjunction, disjunction, etc.) can be performed in polynomial time by Bryant's APPLY algorithm. If two EQ-OBDDs are combined in this way, new substitutions must be done in both of them, which destroy the ordering. We can of course re-order them by using the rewrite system ORDER, but the advantage of having a polynomial APPLY has been lost.

As an alternative, we use a top-down approach, which in the context of OBDDs has for instance been described in [12]. This approach is based on the Shannon expansion. For propositional logic, this reads: $\Phi \iff \mathrm{ITE}(p, \Phi|_p, \Phi|_{\neg p})$, where in $\Phi|_p$ all occurrences of p are replaced by $\mathbf{1}$, and in $\Phi|_{\neg p}$ by $\mathbf{0}$. Taking for p the smallest propositional variable in the ordering, this Shannon expansion can be used to create a root node for p, and recursively continuing with two subformulae that do not contain p. The number of variables in the formula decreases. So, this process terminates. Because at each step the smallest variable is taken, the resulting BDD is ordered.

When p is an equation, say $x = y$, the Shannon expansion still holds. In the formula $\Phi|_{x=y}$, we assume that $x = y$, so we are allowed to substitute y for x. This leads to the following variant of the Shannon expansion:

$$\Phi \iff \mathrm{ITE}(x = y, \Phi[x := y], \Phi[(x = y) := \mathbf{0}])$$

This is recursively applied, with $x = y$ the smallest equation in Φ, oriented in such a way that $x \prec y$ in the variable order. Due to the substitutions it is not guaranteed that the resulting EQ-BDD is ordered. However, we will show that repeatedly applying the Shannon expansion does lead to an EQ-OBDD.

3.1 A Topdown Algorithm

We now describe the algorithm precisely. We introduce a term rewrite system SIMPLIFY, which removes superfluous occurrences of $\mathbf{0}$ and $\mathbf{1}$ and orients all guards. It is clearly terminating and confluent.

Definition 15. *The TRS* SIMPLIFY *consists of the following rules:*

$$\begin{array}{llll} \mathbf{0} \wedge T \to \mathbf{0} & \neg\mathbf{1} & \to \mathbf{0} & \\ T \wedge \mathbf{0} \to \mathbf{0} & \neg\mathbf{0} & \to \mathbf{1} & x = x \to \mathbf{1} \\ \mathbf{1} \wedge T \to T & \mathrm{ITE}(\mathbf{1}, T, U) & \to T & y = x \to x = y \quad \text{if } x \prec y \\ T \wedge \mathbf{1} \to T & \mathrm{ITE}(\mathbf{0}, T, U) & \to U & \end{array}$$

We write $\Phi\!\downarrow$ for the normal form of Φ obtained by this rewrite system. Φ is called simplified, if $\Phi \equiv \Phi\!\downarrow$.

Note that every closed formula rewrites to $\mathbf{0}$ or $\mathbf{1}$. Furthermore, on EQ-BDDs only the last four rules are applicable. Finally, note that ordered EQ-BDDs are simplified. We introduce an auxiliary operation $\Phi|_s$, where Φ is a formula and s a guard or the negation of a guard. We assume that Φ is simplified.

Definition 16. *We define $\Phi|_s$, where s is p, $\neg p$, $x = y$ or $x \neq y$ as follows: If $s \equiv p$, then $\Phi|_s$ consists of replacing all occurrences of p by $\mathbf{1}$; in $\Phi|_{\neg s}$ all occurrences of p are replaced by $\mathbf{0}$. In case $s \equiv x = y$, we obtain $\Phi|_s$ by replacing all occurrences of y by x, and $\Phi|_{\neg s}$ by replacing $x = y$ by $\mathbf{0}$ everywhere.*

Example 17. Let $\Phi \equiv x = z \wedge y = z$ and $g \equiv x = z$ and assume $x \prec y \prec z$. Then $\Phi|_g \equiv x = x \wedge y = x$ and $\Phi|_{\neg g} \equiv \mathbf{0} \wedge y = z$. After simplification, we get: $\Phi|_g\!\downarrow \equiv x = y$ and $\Phi|_{\neg g}\!\downarrow \equiv \mathbf{0}$. □

We are now ready to define the basic top-down transformation algorithm:

Definition 18. *Assume that Φ be a simplified formula. We define the algorithm* TOPDOWN *on input Φ as follows:*

- $\textsc{Topdown}(\mathbf{1}) \equiv \mathbf{1}$
- $\textsc{Topdown}(\mathbf{0}) \equiv \mathbf{0}$
- *Otherwise, let g be the smallest guard occurring in Φ. Then*

$$\textsc{Topdown}(\Phi) \equiv \overline{\mathrm{ITE}}(g, \textsc{Topdown}(\Phi|_g\downarrow), \textsc{Topdown}(\Phi|_{\neg g}\downarrow))$$

where

$$\overline{\mathrm{ITE}}(g,T,U) \equiv \begin{cases} T & \text{if } T \equiv U \\ \mathrm{ITE}(g,T,U) & \text{otherwise.} \end{cases}$$

Note that a closed formula simplifies to $\mathbf{1}$ or $\mathbf{0}$, so in the other case it must contain a guard. Note that due to substitutions, new equalities can be introduced on the fly. We now prove termination and soundness of the algorithm TOPDOWN. With $\#(\Phi)$ we denote the number of guard occurrences in the completely unfolded tree of Φ. Note that none of the rules from SIMPLIFY increases the number of guards, so we have the following:

Lemma 19. *For any formula Φ, we have $\#(\Phi) \geq \#(\Phi\downarrow)$.*

Lemma 20. *Let Φ be a simplified formula, and let g be a simplified guard.*

(1) $\#(\Phi) \geq \#(\Phi|_g)$ (3) *if g occurs in Φ, then* $\#(\Phi) > \#(\Phi|_g)$
(2) $\#(\Phi) \geq \#(\Phi|_{\neg g})$ (4) *if g occurs in Φ, then* $\#(\Phi) > \#(\Phi|_{\neg g})$

Proof. Simultaneous formula induction on Φ. This boils down to checking that in Definition 16, each guard is replaced by at most one other guard. □

Theorem 21. *The algorithm* $\textsc{Topdown}(\Phi)$ *always terminates.*

Proof. With each recursive call, $\#(\Phi)$ strictly decreases. □

Theorem 22 (soundness). *For any formula Φ, we have:* $\Phi \iff \textsc{Topdown}(\Phi)$

Proof. Induction over the number of calls to TOPDOWN. The induction step uses that $\Phi \iff \Phi\downarrow$ and $g \Rightarrow (\Phi \iff \Phi|_g)$ and similar for $\neg g$. □

3.2 Iteration of TOPDOWN

Unfortunately, it is *not* the case that $\textsc{Topdown}(\Phi)$ is always ordered, as the following example shows.

Example 23. Assume $x \prec y \prec z$. Then $\textsc{Topdown}(x \neq y \wedge (x = z \wedge y = z)) \equiv \mathrm{ITE}(x = y, \mathbf{0}, \mathrm{ITE}(x = z, \mathrm{ITE}(x = y, \mathbf{1}, \mathbf{0}), \mathbf{0}))$. See Figure 2, where the formulae in square brackets denote the arguments to TOPDOWN, and the dashed nodes occur in the call graph, but are suppressed in the resulting EQ-BDD. In the low branch, $x = y$ is replaced by $\mathbf{0}$, but due to substitutions in the recursive call, new occurrences of $x = y$ are generated. Note that this is dangerous, as after one application of TOPDOWN it still contains unsatisfiable paths, which erroneously could lead one to believe that the EQ-BDD represents a satisfiable formula. □

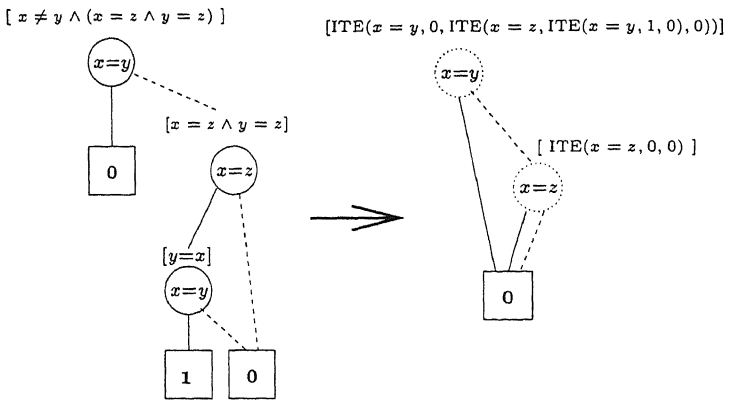


Fig. 2. Two call-graphs to TOPDOWN.

Note that in the previous example, an EQ-OBDD is found by another application of TOPDOWN. We propose to apply TOPDOWN repeatedly to a formula Φ, until a fixed point is reached. In the benchmarks presented in Section 3.3 at most two iterations of TOPDOWN were required to obtain an EQ-OBDD. In the rest of this section we prove that the fixed point can be reached in a finite number of steps, and that it is an ordered EQ-BDD.

Lemma 24. *Let Φ be a simplified EQ-BDD and g be a simplified guard. Then*

(1) $\Phi \succeq_{\text{rpo}} \Phi|_g\downarrow$ (3) *if g occurs in Φ, then* $\Phi \succ_{\text{rpo}} \Phi|_g\downarrow$
(2) $\Phi \succeq_{\text{rpo}} \Phi|_{\neg g}\downarrow$ (4) *if g occurs in Φ, then* $\Phi \succ_{\text{rpo}} \Phi|_{\neg g}\downarrow$

Proof. We apply simultaneous induction on the structure of Φ. We only present two interesting fragments of the proof of case (1) and (3), where $\Phi \equiv \text{ITE}(u = v, T, U)$ and $g \equiv x = y$. Note that $x \prec y$ and $u \prec v$, because Φ and g are simplified.

First consider case (1). By definition $\Phi|_g\downarrow \equiv \text{ITE}((u = v)|_g\downarrow, T|_g\downarrow, U|_g\downarrow)\downarrow$. Observe that $(u = v)|_g\downarrow$ either equals $\mathbf{1}$, $x = v$ (if $u \equiv y$), $u = x$ (if $v \equiv y$ and $u \prec x$), $x = u$ (if $v \equiv y$ and $x \prec u$) or $u = v$. The case $v = x$ does not occur, for we would have $v \prec x \prec y \equiv u \prec v$.

In the first case $\Phi|_g\downarrow \equiv T|_g\downarrow$. Using the induction hypothesis, $T \succeq_{\text{rpo}} T|_g\downarrow$. By property (I) of recursive path orderings it follows that $\Phi \succ_{\text{rpo}} T$ and hence $\Phi \succ_{\text{rpo}} \Phi|_g\downarrow$. In the next three cases, it is obvious that $x = v \prec u = v$ and $u = x \prec u = v$ and $x = u \prec u = v$, respectively. Now using a similar argument as above, we can show that $\Phi \succ_{\text{rpo}} T|_g\downarrow$ and $\Phi \succ_{\text{rpo}} U|_g\downarrow$. So, by property (II) of RPO it follows that $\Phi \succ_{\text{rpo}} \Phi|_g\downarrow$. In the last case, where $(u = v)|_g\downarrow \equiv u = v$, we find by the induction hypothesis $T \succeq_{\text{rpo}} T|_g\downarrow$ and $U \succeq_{\text{rpo}} U|_g\downarrow$. By property (III) of RPO it follows that $\Phi \succeq_{\text{rpo}} \Phi|_g\downarrow$.

Now consider case (3). Note that in case (1) we proved that $\Phi \succ_{\text{rpo}} \Phi|_g\downarrow$ in all but the case where $(u = v)|_g\downarrow \equiv u = v$. So, we only need to consider this

case. As g occurs in Φ, it must occur in T or in U. As the cases are symmetric, we can without loss of generality assume that g occurs in T. Via the induction hypothesis it follows that $T \succ_{\text{rpo}} T|_g\downarrow$. Furthermore, by case (1) $U \succeq_{\text{rpo}} U|_g\downarrow$. So, by property (III) of RPO we can conclude that

$$\Phi \equiv \text{ITE}(u = v, T, U) \succ_{\text{rpo}} \text{ITE}(u = v, T|_g\downarrow, U|_g\downarrow) \equiv \Phi|_g\downarrow. \qquad \square$$

Lemma 25. *Let Φ be a simplified EQ-BDD.*

1. $\Phi \succeq_{\text{rpo}} \textsc{Topdown}(\Phi)$.
2. *Φ is ordered iff* $\Phi \equiv \textsc{Topdown}(\Phi)$.

Proof. Part 1 is proved by induction on $\#(\Phi)$. Note that if Φ does not contain a guard then it is equal to **1** or **0**, and this theorem is trivial. So, assume Φ contains at least one guard and let g be the smallest guard occurring in Φ. Recall from Lemma 19, 20 that $\#(\Phi) > \#(\Phi|_g\downarrow)$ and similar for $\neg g$. Then $\textsc{Topdown}(\Phi) = \overline{\text{ITE}}(g, \textsc{Topdown}(\Phi|_g\downarrow), \textsc{Topdown}(\Phi|_{\neg g}\downarrow))$. By induction hypothesis and Lemma 24, we have:

$$(*) \qquad \begin{array}{c} \Phi \succ_{\text{rpo}} \Phi|_g\downarrow \succeq_{\text{rpo}} \textsc{Topdown}(\Phi|_g\downarrow) \\ \Phi \succ_{\text{rpo}} \Phi|_{\neg g}\downarrow \succeq_{\text{rpo}} \textsc{Topdown}(\Phi|_{\neg g}\downarrow) \end{array}$$

First, assume $\textsc{Topdown}(\Phi|_g\downarrow) \equiv \textsc{Topdown}(\Phi|_{\neg g}\downarrow)$. Then $\textsc{Topdown}(\Phi) \equiv \textsc{Topdown}(\Phi|_g\downarrow)$ and we are done by (*). Now assume $\textsc{Topdown}(\Phi|_g\downarrow) \not\equiv \textsc{Topdown}(\Phi|_{\neg g}\downarrow)$, and assume that $\Phi \equiv \text{ITE}(h, T, U)$. Then $\textsc{Topdown}(\Phi) \equiv \text{ITE}(g, \textsc{Topdown}(\Phi|_g\downarrow), \textsc{Topdown}(\Phi|_{\neg g}\downarrow))$. As g is the smallest guard, one of the following two cases must hold.

- $g \equiv h$. In this case $\Phi|_g\downarrow \equiv T|_g\downarrow$. Using Lemma 24 and the induction hypothesis, we can conclude $T \succeq_{\text{rpo}} T|_g\downarrow \equiv \Phi|_g\downarrow \succeq_{\text{rpo}} \textsc{Topdown}(\Phi|_g\downarrow)$. Similarly, $U \succeq_{\text{rpo}} \textsc{Topdown}(\Phi|_{\neg g}\downarrow)$. By case (III) of RPO it follows that $\Phi \succeq_{\text{rpo}} \textsc{Topdown}(\Phi)$.
- $h \succ g$. Using (*) we can immediately apply case (II) of RPO and conclude that $\Phi \succ_{\text{rpo}} \textsc{Topdown}(\Phi)$.

Part 2. Both directions are proved by structural induction on Φ. $\Longrightarrow$: We must show that if Φ is ordered, then $\Phi \equiv \textsc{Topdown}(\Phi)$. The case where Φ equals **0** or **1** is trivial. So, consider the case where $\Phi \equiv \text{ITE}(g, T, U)$. As Φ is ordered, g must be the smallest guard of Φ and cannot occur in T or U. Also, if $g \equiv x = y$, y does not occur in T. Moreover, T and U are ordered, hence also simplified. So, $\Phi|_g\downarrow \equiv T$ and $\Phi|_{\neg g}\downarrow \equiv U$. Note that $T \not\equiv U$.

$$\begin{array}{ll} \textsc{Topdown}(\Phi) \equiv & \\ \overline{\text{ITE}}(g, \textsc{Topdown}(\Phi|_g\downarrow), \textsc{Topdown}(\Phi|_{\neg g}\downarrow)) \equiv & \\ \overline{\text{ITE}}(g, \textsc{Topdown}(T), \textsc{Topdown}(U)) \equiv & \text{(Induction hypothesis)} \\ \text{ITE}(g, T, U) \equiv & \\ \Phi & \end{array}$$

$\Longleftarrow$: Assume $\Phi = \textsc{Topdown}(\Phi)$. If Φ is **1** or **0** then it is trivially ordered. So assume $\Phi = \text{ITE}(g, \Phi_1, \Phi_2)$. Then $\textsc{Topdown}(\Phi) = \overline{\text{ITE}}(h, \Psi_1, \Psi_2)$, where h is

the smallest guard in Φ, $\Psi_1 \equiv \text{TOPDOWN}(\Phi|_h\downarrow)$ and $\Psi_2 \equiv \text{TOPDOWN}(\Phi|_{\neg h}\downarrow)$. If $\Psi_1 \equiv \Psi_2$, then $\Phi \equiv \Psi_1$ and using Lemma 24.3 and 25.1 we get the following contradiction: $\Phi \succ_{\text{rpo}} \Phi|_h\downarrow \succeq_{\text{rpo}} \Psi_1 \equiv \Phi$.

Hence $\Psi_1 \not\equiv \Psi_2$. Then it must be the case that $g \equiv h$, $\Phi_1 \equiv \Psi_1$ and $\Phi_2 \equiv \Psi_2$. Note that then $\Phi|_h\downarrow \equiv \Phi_1|_h\downarrow$. Now, as $\Phi_1 \succeq_{\text{rpo}} \Phi_1|_h\downarrow \equiv \Phi|_h\downarrow \succeq_{\text{rpo}} \Psi_1$ it must be the case that $\Phi_1 \equiv \Phi_1|_h\downarrow$, hence $\Phi_1 \equiv \text{TOPDOWN}(\Phi_1)$. Similarly, $\Phi_2 \equiv \text{TOPDOWN}(\Phi_2)$.

We must show that Φ is ordered. By induction hypothesis, Φ_1 and Φ_2 are ordered, so no rule of the TRS ORDER is applicable to a strict subterm of Φ. We now show that no rule (1–8) is applicable to the root of Φ:

If rule 1 is applicable, then $\Psi_1 \equiv \Psi_2$, which we excluded already. In case of rule 2, $\Phi_1 \equiv \text{ITE}(g, T, U)$, and we obtain the following contradiction: $\Phi_1 \succ_{\text{rpo}} T \succeq_{\text{rpo}} T|_g\downarrow \equiv \Phi_1|_g\downarrow \equiv \Phi_1$. Rule 3 is excluded similarly. Rule 4 and 5 are not applicable because $g \equiv h$, which is the smallest guard in Φ. Rule 6 and 7 are not applicable because Φ is simplified. Finally, if rule 8 were applicable, $g \equiv x = y$ and y occurs in Φ_1. Then, using monotonicity of $\succ_{\text{rpo}}$, we have the following contradiction: $\Phi_1 \succ_{\text{rpo}} \Phi_1[y := x] \equiv \Phi_1|_g \succeq_{\text{rpo}} \Phi_1|_g\downarrow \equiv \Phi_1$. The last inequality uses the fact that the applicable rules of SIMPLIFY are contained in $\succ_{\text{rpo}}$. □

Theorem 26. *Let Φ be a simplified formula. Iterated application of* TOPDOWN *to Φ leads in a finite number of steps to an EQ-OBDD equivalent to Φ.*

Proof. After one application of TOPDOWN, Φ is transformed into a simplified EQ-BDD. So, iterated application of TOPDOWN leads to a sequence $\Phi, \Phi_1, \Phi_2, \ldots$ of which each Φ_i ($i \geq 1$) is a simplified EQ-BDD. By Lemma 25.1 the sequence $\Phi_1, \Phi_2, \ldots$ is decreasing in a well-founded way. Hence, at a certain point in the sequence we find that $\Phi_i \equiv \Phi_{i+1}$. By Lemma 25.2 Φ_i is the required EQ-OBDD. Note that by Lemma 25.2 Φ_i is the first ordered EQ-BDD in the sequence. □

We conclude with the complete algorithm to transform an arbitrary formula Φ to EQ-OBDD, which is just a repeated application of TOPDOWN until a fixed point is reached:

$$\texttt{EQ-OBDD}(\Phi) = \texttt{fixedpoint}(\text{TOPDOWN})(\Phi\downarrow)$$

We stress that in the benchmarks we never needed more than 2 iterations. This is not generally the case:

Example 27. Given $a \prec b \prec c \prec d \prec e \prec f$, the following EQ-BDD needs 4 iterations: $\text{ITE}(a = f, \text{ITE}(a = e, d = e, c = d), b = c)$. The intermediate EQ-BDDs have size 9, 13, 23 and 21, respectively. This can be checked with our implementation. □

3.3 Implementation and Benchmarks

In order to study the performance of TOPDOWN, we made an implementation and used it to try the benchmarks reported in [16, 19]. The authors report to have comparable performance as in [10]. Unfortunately, we could not obtain the benchmarks used in [7]. We first describe the implementation, including some variable orderings we used and then present the results.

Prototype implementation. We have made a prototype implementation of the TOPDOWN algorithm. As programming language we used C, including the `ATerm`-library [4]. The basic data types in this library are `ATerms` and `ATermTables`. `ATerms` are terms, which are internally represented as maximally shared DAGs. As a consequence, syntactical equality of terms can be tested in constant time. The basic operations are term formation and decomposition, which are also performed in constant time. `ATermTables` implement hash tables of dynamic size, with the usual operations. The `ATerm`-library also provides memory management functionality, by automatically garbage collecting unreferenced terms. By representing formulae and BDDs as `ATerms`, we are sure that they are always a maximally shared DAG.

Care has to be taken in order to avoid that during some computation, shared subterms are processed more than once. Therefore all recursive procedures, like "find the smallest variable", "simplify" and $\Phi|_s$ are implemented using a hash table to implement memoization. In this way, syntactically equal terms are processed only once, and the time complexity for computing these functions is linear in the number of nodes in the DAG, which is the number of *different* subterms in the formulae. Also the TOPDOWN-function itself uses a hash table for memoization. This contributes to its efficiency: Consider a formula Ψ which is symmetric in p and q (for instance: $(p \wedge q) \vee \Phi$, or $(p \wedge \Phi) \vee (q \wedge \Phi)$). Then $(\Psi|_p\downarrow)|_{\neg q}\downarrow \equiv (\Psi|_{\neg p}\downarrow)|_q\downarrow$. Thanks to memoization, only one of them will actually be computed. Still, the TOPDOWN function has worst case exponential behavior, which is unavoidable, because in the propositional case (i.e. excluding equations) it builds an OBDD from a propositional formula in one iteration. Due to memoization of TOPDOWN's arguments, the memory demands are rather high.

Results. Benchmark formulae can be obtained from [19] and most of them could be solved with the methods described in [16]. Each formula is known to be a tautology. They originate from compiler optimization; each formula expresses that the source and target code of a compilation step are equivalent. We used the versions where Ackermann's function elimination has been applied [1], but domain minimization [16] has not yet been applied. In fact, our method does not rely on the finiteness of domains at all. The benchmark formulae extend the formulae of Definition 1 in various ways, but these extensions could be dealt with easily.

It is well known that the variable ordering has an important effect on the performance. We therefore tried a number of orderings: With 't' we denote the textual order of the variables as given in [19]. With 'r' we denote the reverse of this textual order. Finally, 'bt' ('br') denotes the textual (reverse) order, except that boolean variables always precede domain variables.

We can now present the results. They can be found in Figure 3. The first column contains the number of the files, as given in [19]. The next three columns give an indication of the size of the formula: #b is the number of boolean variables, #d the number of domain variables, and #n is the number of nodes in a maximally shared representation of the formula. The fifth column contains the

Nr. file	#d	#b	#n	[16, 19]	t	bt	r	br
022	59	49	993	:0.16	:13	:16	17:01	7:50
025	45	55	285	:0.2	:0.3	:0.3	:0.1	:0.1
027	21	60	569	:1.7	12:37	10:55	—	—
032	16	48	525	:0.1	:3.2	:3.2	5:02	4:12
037	12	26	942	:0.15	2:17	:2.3	7:28	:12
038	6	14	844	:0.18	:17	:0.4	:6.8	:0.3
043	158	72	1717	—	—	—	—	—
044	39	14	383	:0.1	:3.7	:2.0	0:28	:1.6
046	68	35	667	:0.13	—	—	—	—
049	163	75	1717	—	—	—	:0.3	:0.1

Fig. 3. Timing results for the benchmarks

times reported in [19], obtained by the method of [16]. The other columns show our results, using various variable orderings. Each entry is in minutes, i.e. $a : b.c$ means a minutes, and $b.c$ seconds. With — we denote that a particular instance could not be solved, due to lack of memory. The times are including the time to start the executable, I/O and transforming the benchmarks to the `ATerm` format. We used an IRIX machine with 300 MHz and where the processes could use up to 1.5 GB internal memory.

The table shows that we can solve 8 out of 10 formulae. In this respect our method is comparable to [16]. The exact times are not relevant, because we have made a prototype implementation, without incorporating all well-known optimizations applied in BDD-packages, whereas [19] used an existing BDD-package.

It is also clear that the variable ordering is rather important. In most cases, it is a good idea to split on boolean variables first, before splitting on equalities. The reason probably is that splitting on an equality introduces new guards, which can be rather costly. We also counted the number of iterations of TOPDOWN that were needed in order to reach an EQ-OBDD. Remarkably, the maximum number of iterations was 2 and nearly all time was spent in the first iteration. Most benchmarks even reached a fixed point in the first iteration.

We conclude that the algorithm TOPDOWN is feasible. This is quite remarkable, as the top-down method is usually regarded as inefficient. We attribute this to the use of maximal sharing and memoization. In the next standard example, it is even more effective than using APPLY.

Example 28. Consider the formula $X \equiv p \wedge (\Phi \wedge \neg p)$. In case p is the smallest variable, TOPDOWN terminates in one call, because $X|_p\downarrow \equiv \mathbf{0}$ and $X|_{\neg p}\downarrow \equiv \mathbf{0}$ and a contradiction is detected. □

The usual APPLY algorithm will completely build the tree for Φ, potentially resulting in an exponential blow-up. Many heuristics for providing a variable ordering will make p minimal, so this is a realistic scenario. In [2] an adaptation

to the original APPLY algorithm is described, which also solves this formula in constant time.

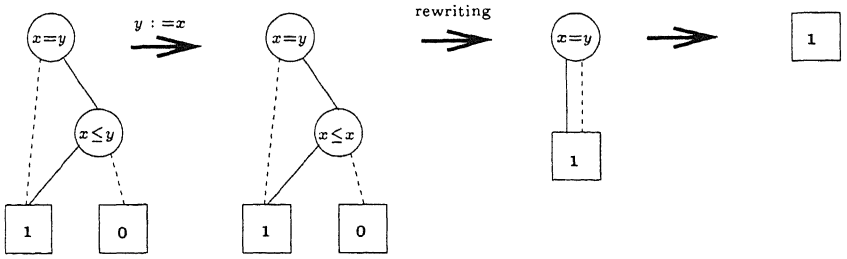
4 Future Work

Our motivation originates from investigations in the computer-aided analysis of distributed systems and protocols, where data is usually specified by algebraic data types, and automated reasoning is generally based on term rewriting. For this reason, function symbols cannot be eliminated, and the domains are generally structured and often infinite. For instance, as soon as we introduce the successor function on natural numbers, all interesting models are infinite.

Our approach forms an extendible basis. We may allow function symbols in EQ-BDDs. In the algorithm, the rewrite rules of the data domain can be added to the TRS SIMPLIFY. In this way, one is able to prove for instance that $x \leq y \vee x \neq y$ is a tautology. Obviously this is not true when the interpretation of functions is free (e.g. interpret $\leq$ as $<$). However, consider the following definition of $\leq$ in terms of rewrite rules, where S denotes the successor function:

$$x < 0 \to \mathbf{0} \qquad x < S(y) \to x \leq y \qquad x \leq y \to x < y \vee x = y$$

An EQ-OBDD proof with auxiliary rewrite rules of $x \leq y \vee x \neq y$ looks as follows:



Also, $x \leq 0 \wedge y = 0 \to x = y$ can be proved in this way. Note that this doesn't hold on the integers or reals, so the logic of difference constraints [13] cannot be used here.

As future work we plan to investigate under which conditions such extensions are complete. For instance, in the example above we at least additionally need the following rules:

$$0 = S(x) \to \mathbf{0} \qquad S(x) = S(y) \to x = y \qquad x = S^n(x) \to \mathbf{0}.$$

We also plan to improve and extend our algorithm in the presence of function symbols. One of the main issues here is how to extend the ordering on the new nodes.

Acknowledgments. We like to thank Ofer Shtrichman for making his benchmarks publicly available and discussing them. We are also indebted to the anonymous referees for improving some of the proofs.

References

1. W. Ackermann. *Solvable Cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. H. R. Andersen and H. Hulgaard. Boolean expression diagrams. In *Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 88–98, Warsaw, Poland, 1997. IEEE Computer Society.
3. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
4. M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software - Practice & Experience*, 30:259–291, 2000. See also `http://www.cwi.nl/projects/MetaEnv/aterm/`.
5. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
6. R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
7. R.E. Bryant, S. German, and M.N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In N. Halbwachs and D Peled, editors, *Proc. of Computer Aided Verification, CAV'99*, LNCS 1633. Springer-Verlag, 1999.
8. Burch, J.R. and D.L. Dill. Automatic verification of pipelined microprocessors control. In D. L. Dill, editor, *Proceedings of Computer Aided Verification, CAV'94*, LNCS 818, pages 68–80. Springer-Verlag, June 1994.
9. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1-2):69–115, 1987.
10. A. Goel, K. Sajid, H. Zhou, and A. Aziz. BDD based procedures for a theory of equality with uninterpreted functions. In *Proceedings of Computer Aided Verification, CAV'98*, LNCS 1427, pages 244–255. Springer-Verlag, 1998.
11. J.W. Klop. Term rewriting systems. In D. Gabbay S. Abramski and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1. Oxford University Press, 1991.
12. C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design: OBDD-Foundations and Applications.* Springer-Verlag, 1998.
13. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In J. Flum and M. Rodriguez-Artalejo, editors, *Computer Science Logic (CSL'99)*, LNCS 1683. Springer-Verlag, 1999.
14. G. Nelson and D.C. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, 1980.
15. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Proceedings of Computer Aided Verification CAV'96*, LNCS 1102, pages 411–414. Springer-Verlag, 1996.
16. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. In N. Halbwachs and D Peled, editors, *Proceedings of Computer Aided Verification, CAV'99*, LNCS 1633. Springer-Verlag, 1999.
17. J.C. van de Pol and H. Zantema. Binary decision diagrams by shared rewriting. In M. Nielsen and B. Rovan, editors, *Proceedings of Mathematical Foundations of Computer Science, MFCS'00*, LNCS 1893. Springer-Verlag, 2000.
18. R.E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, 1978.
19. O. Shtrichman. Benchmarks for satisfiability checking of equality formulas. See `http://www.wisdom.weizmann.ac.il/~ofers/sat/bench.htm`, 1999.