Equational Binary Decision Diagrams

Jan Friso Groote

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Department of Mathematics and Computing Science, Eindhoven University of Technology

P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Email: JanFriso.Groote@cwi.nl

Jaco van de Pol

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Email: Jaco.van.de.Pol@cwi.nl

Abstract

We incorporate equations in binary decision diagrams (BDD). The resulting objects are called EQ-BDDs. A straightforward notion of 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 for 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 is proved to be correct and terminating, by means of recursive path ordering. The algorithm has been implemented, and applied to benchmarks known from literature. The performance of a prototype implementation is comparable to existing proposals.

CR Subject Classification (1991): F.2, F.4
AMS Subject Classification (1991): 03B10, 03B20, 03B35, 03B70, 68T15
Keywords & Phrases: Binary Decision Diagrams, Equality

1 Introduction

Motivation and background. The correctness of hardware designs can be formally expressed in propositional logic. For scaling up the verification of such hardware correctness formulae, it appears to be useful to extend propositional logic with *uninterpreted functions* over arbitrary domains, and *equality* (=) on these domains [14]. Parts of the hardware design that are not essential for the verification can be abstracted from, by replacing them by a function symbol. Equality is used for instance to express equivalence of specification and design. Now the task is to check satisfiability (or tautology) of formulae of propositional logic with equality and uninterpreted function symbols (EUF). Such a method usually proceeds in three steps:

1. Elimination of function symbols

2. Reduction to propositional logic

3. Check with existing BDD-package

Ad 1. By a result due to Ackermann [1], the function symbols can be eliminated, at the cost of introducing new variables and congruence constraints. In essence, subterms like $F(x)$ and $F(y)$ are replaced by new variables f_1 and f_2, and the the functionality constraint $x = y \rightarrow f_1 = f_2$ is added. This yields a formula of propositional logic with equality, which is satisfiable if, and only if, the original formula is.

Ad 2. Such formulae have the *finite domain property*, which means that they are satisfiable if, and only if, they are satisfiable in a suitably large finite model; the number of different variables is an upper bound. Then, a variable over a domain of size n can be replaced by $\lceil \log n \rceil$ fresh boolean variables.

Ad 3. Checking satisfiability of propositional formulae is often performed using Binary Decision Diagrams (BDDs) [6] (see also [7, 16]). In an *ordered* BDD (OBDD) a strict order on the variables is imposed. The resulting data structure yields unique representations for boolean functions, which are quite compact for large classes of formulae. By uniquess, checking tautology, contradiction or satisfiability of the resulting OBDD can be performed in constant time.

Recent contributions. Three recent papers [11, 8, 17] refine the aproach mentioned above in various directions. Ackermann's reduction (Step 1) is improved by Bryant et al. [8] in the following way: In order to avoid the functionality constraints, the subterms $F(x)$ and $F(y)$ are replaced by f_1 and the if-then-else term $\mathrm{ITE}(x = y, f_1, f_2)$, respectively. Functionality is now built in automatically. Their main contribution, however, 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. This technique reduces the number of boolean variables obtained by step 2, the reduction to propositional logic.

Pnueli et al. [17] use Ackermann's reduction (step 1) and improve step 2, by providing heuristics to obtain lower estimates for the domains. These estimates are obtained by taking the structure of the formula into account. Their major case distinction is also between positive and negative occurrences of equations.

Goel et al. [11] improve step 2, by avoiding bit vectors for finite domains at all. Instead, they introduce boolean variables e_{ij}, representing the equation $x_i = x_j$. In fact, this method does not rely on the finite model property. However, the resulting BDD has to be traversed with care. A satisfying interpretation in the BDD might violate transitivity constraints of the form $e_{ij} \wedge e_{jk} \rightarrow e_{ik}$. The question whether an OBDD has a transitivity-consistent satisfaction is proved to be NP-complete.

In a technical report, Bryant et al. [9] improve on the latter method by predicting which transitivity instances might be needed. Only these are added to the propositional formula before the BDD is built. The distinction between positive and negative equations is beneficial again, for in the special case they studied, most transitivity constraints appear to correspond with p-symbols, which have been replaced by fixed bit patterns.

The similarity between all these methods is that they reduce the original EUF formula to propositional logic, and then use an existing BDD package to check satisfiability. All aproaches have a blow-up especially when equations occur negatively. This blow-up is caused because either the domains, or the number of transitivity constraints get large.

Our approach. We introduce EQ-BDDs, which are BDDs whose internal nodes may contain equations between variables (similar to the e_{ij} variables). In this way, step 2 is avoided completely and the equalities are maintained, at the expense of generalizing the BDD theory and reconstructing a BDD-package.

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 an ordered EQ-BDD (EQ-OBDD) of the form $\mathrm{ITE}(x = y, P, Q)$, y may not occur in P, but should be substituted 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. However, we show that in an EQ-OBDD, each path from the root to a leaf is consistent. As a corollary, $\mathbf{0}$ is the only contradictory EQ-OBDD, and $\mathbf{1}$ 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, as opposed to [11], where transitivity violations have to be taken into account.

We present an algorithm for converting propositional formulae (or circuits) containing equations into an EQ-OBDD. We were not able to find an efficient generalization of the usual bottom-up algorithm, where logical operations are repeatedly applied to already constructed OBDDs. Each operation can be performed in polynomial time, Bryant's Apply algorithm [6], which runs in quadratic time. This yields relatively effective procedures to transform a propositional logical formula into an OBDD.

Instead, we use a generalization of the top-down method (cf. [16]), which is based on repeated application of Shannon's expansion with the "smallest" equation $x = y$:

$$\Phi \iff \big[(x = y \wedge \Phi|_{x=y}) \vee (x \neq y \wedge \Phi|_{x \neq y})\big],$$

where in $\Phi|_{x=y}$ we replace all occurences of y by x.

The inefficiency usually attributed to this top-down approach is avoided by using memoization techniques and maximal sharing. We have made a prototype implemention in C, which uses the `ATerm` library [5] to manipulate terms in maximally shared representation. We applied this implementation on the benchmarks used in [17, 19] and we exprimented with various variable orderings. It appears that our ideas yield a feasible procedure, and that the performance is comparable to the approach in [17].

Our original motivation for investigating OBDDs with equality is our interest in the verification of distributed programs and protocols. In this setting, functions are generally *interpreted* and domains are often infinite, and have structure. This disallows the use of both Ackermann's function elimination and the finite domain property. In our setting we do not use the finite model property, whereas [8, 17] essentially depend on it. Furthermore, we envisage that function symbols can can straightforwardly be incorporated into EQ-BDDs, both constructor and defined functions. It is not clear to us how to incorporate these in [11, 8, 17]. 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 [18].

The fact that our prototype implementation performs comparably well as existing proposals indicates that extendibility does not necessarily come with a loss in efficiency.

2 EQ-BDDs

Our aim is to check satisfiability and tautology of propositional formulae with equality. In this paper, we assume that function symbols have been eliminated, for instance with Ackermann's function elimination [1]. We now define a syntax for formulae. First assume a set P of proposition (boolean) variables (typically p, q, ...) and a set V of 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 $\neg(x = y)$ and $\Psi \vee \Phi$ as an abbreviation of $\neg(\neg\Phi \wedge \neg\Psi)$. Here $\mathrm{ITE}(\Phi, \Psi, \Xi)$ is called an if-then-else formula. It is equivalent to $(\Phi \wedge \Psi) \vee (\neg\Phi \wedge \Xi)$. In order to

avoid confusion, we write $\equiv$ for syntactic equality, for instance $x \equiv y$ means that x and y are the same variable. It is easy to extend the syntax with other connectives, but they can be handled as the existing ones and only add notational complexity.

The semantics is standard. Given a domain for the variables D, and interpretation functions $I : V \to D$ and $J : P \to \{\mathbf{0}, \mathbf{1}\}$, a formula Φ evaluates to either $\mathbf{0}$ (False) or $\mathbf{1}$ (True); this is denoted by $[B]^I_J$, and can straightforwardly be defined by induction over the syntactic structure of Φ. So a formula can be seen as a representation of a predicate on its input variables. Note that different formulae may be logically equivalent, in the sense that they represent the same predicate.

Definition 2

$$
\begin{aligned}
[\mathbf{0}]^I_J &= \mathbf{0} \\
[\mathbf{1}]^I_J &= \mathbf{1} \\
[p]^I_J &= J(p) \\
[x = y]^I_J &= \begin{cases} \mathbf{1}, \textit{ if } J(x) = J(y) \\ \mathbf{0}, \textit{ otherwise} \end{cases} \\
[\neg\Phi]^I_J &= \begin{cases} \mathbf{1}, \textit{ if } [\Phi]^I_J = \mathbf{0} \\ \mathbf{0}, \textit{ otherwise} \end{cases} \\
[\Phi \wedge \Psi]^I_J &= \begin{cases} \mathbf{1}, \textit{ if } [\Phi]^I_J = \mathbf{1} \textit{ and } [\Psi]^I_J = \mathbf{1} \\ \mathbf{0}, \textit{ otherwise} \end{cases} \\
[\mathrm{ITE}(\Phi, \Psi, \Xi)]^I_J &= \begin{cases} [\Psi]^I_J, \textit{ if } [\Phi]^I_J \\ [\Xi]^I_J, \textit{ otherwise} \end{cases}
\end{aligned}
$$

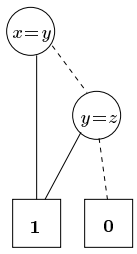
A formula Φ is *valid* in D, if $[\Phi]^I_J = \mathbf{1}$ for all $I : V \to D$ and $J : P \to \{\mathbf{0}, \mathbf{1}\}$. Formula Φ is a *tautology* (also called: universally valid), if it is valid in all domains D. Φ is said to be *satisfiable* if for some domain D and interpretations I, J, $[\Phi]^I_J = \mathbf{1}$. Otherwise it is called contradictory.

Example 3 *Consider the formula* $\mathrm{ITE}(x = y, \mathbf{1}, x = z \vee y = z)$. *It is true when x, y and z range over a domain with two elements, so it is satisfiable. But it is not universally valid. It's negation is satisfiable (although a satisfaction cannot be found in every domain).*

We now turn to the study of EQ-BDDs, which can be seen as a subset of formulae, and turn to arbitrary formulae in Section 3. In the subsequent sections EQ-BDDs and ordered EQ-BDDs are introduced. It is proved that every EQ-BDD is equivalent to an ordered EQ-BDD, and that on the latter the satisfiability check can be done in constant time.

2.1 Ordered EQ-BDDs

A binary decision diagram (BDD [7, 16]) is a DAG, whose internal nodes contain guards, and whose leafs 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 subterm of formulae, i.e. terms, although in implementations we always treat these terms 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 4 *We define the set G of guards and B of EQ-BDDs,*

$$\begin{array}{rcl} G & ::= & P \mid V = V \\ B & ::= & \mathbf{0} \mid \mathbf{1} \mid \mathrm{ITE}(G, B, B) \end{array}$$

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 5 (Order on guards)

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

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 [15, 3]. In particular, a *normal form* is a term to which no rule can be applied. A system is *terminating* if every rewrite sequence is finite.

Definition 6 *A BDD is ordered if, and only if, it is a normal form w.r.t. the following term rewrite system, called* ORDER*:*

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. $\mathrm{ITE}(x = y, T_1[y], T_2) \to \mathrm{ITE}(x = y, T_1[x], T_2)$, *provided* $x \prec y$ *and* y *occurs in* T_1.

The first rule is called the idempotence rule. Rule 1–5 are the standard rules to obtain ordered BDDs, but note that G ranges over propositional variables and equations between domain variables. The usual rule for obtaining maximal sharing is left out, because terms are always regarded as maximally shared DAGs. Rules 6–8 capture the properties of equality, viz. reflexivity, symmetry, and substitutivity. From these rules, transitivity can be derived, as we show in the following example. 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. We could also have stipulated that *at least* or *exactly* one occurrence of y is replaced, as this does not affect the transitive closure of the rewrite relation.

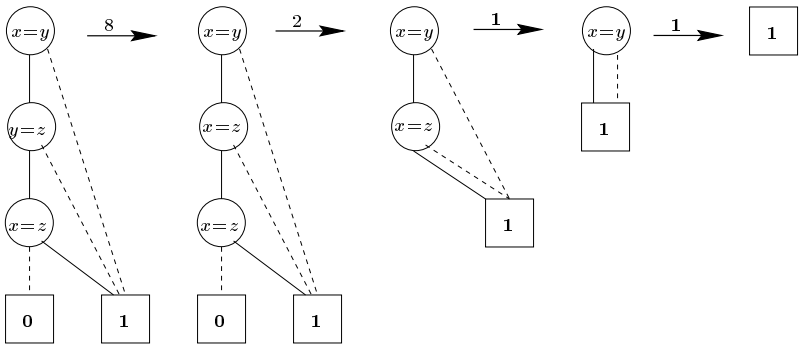


Figure 1: Deriviation of transitivity of equality in EQ-BDDs

Example 7 (Transitivity) *Transitivity is now derived automatically, as the following derivation shows (we assume $x \prec y \prec z$):*

$$\begin{aligned} & \mathrm{ITE}(x=y, \mathrm{ITE}(y=z, \mathrm{ITE}(x=z, \mathbf{1}, \mathbf{0}), \mathbf{1}), \mathbf{1}) \\ \xrightarrow{8} \quad & \mathrm{ITE}(x=y, \mathrm{ITE}(x=z, \mathrm{ITE}(x=z, \mathbf{1}, \mathbf{0}), \mathbf{1}), \mathbf{1}) \\ \xrightarrow{2} \quad & \mathrm{ITE}(x=y, \mathrm{ITE}(x=z, \mathbf{1}, \mathbf{1}), \mathbf{1}) \\ \xrightarrow{1} \quad & \mathrm{ITE}(x=y, \mathbf{1}, \mathbf{1}) \\ \xrightarrow{1} \quad & \mathbf{1} \end{aligned}$$

This rewrite sequence is depicted in Figure 1

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), 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.

2.2 Termination

We have defined EQ-OBDDs as the normal forms of a rewrite system operating on EQ-BDDs. We now show that for each EQ-BDD an equivalent EQ-OBDD exists by showing that the TRS ORDER always terminates. To this end, we use the powerful *recursive path ordering* (RPO) [10, 3]. We use that in EQ-BDDs the first argument of ITE is always a guard. We will apply RPO on trees, where the guards are the internal nodes, and **1** and **0** are the leafs. To emphasize this, we write $g(T, U)$ instead of $\mathrm{ITE}(g, T, U)$ in the termination proof.

RPO needs an ordering on the function symbols. For this we just use the total order on guards of Definition 5, with **1** and **0** added as minimal elements. We use the following definition of the recursive path order:

Definition 8 *$s \equiv f(s_1, s_2) \succ_{\mathrm{rpo}} g(t_1, t_2) \equiv t$ if, and only if, either:*

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

Here $x \succeq_{\mathrm{rpo}} y$ means: $x \succ_{\mathrm{rpo}} y$ or $x \equiv y$. From the literature, it is well known that this definition yields an order (in particular the relation is transitive), which is well-founded and monotone, so it is useful in proving termination.

Lemma 9 *If $x \succ y$, then $T[x] \succ_{\mathrm{rpo}} T[y]$.*

Proof: Induction on T, using monotonicity of $\succ_{\mathrm{rpo}}$. ⊠

Lemma 10 *The rewrite system in Definition 6 is terminating.*

Proof: It suffices to prove that each rewrite rule is contained in $\succ_{\mathrm{rpo}}$.

1. $g(T_1, T_1) \succ_{\mathrm{rpo}} T_1$ by (I).
2. $g(T_1, T_2) \succ_{\mathrm{rpo}} T_1$ by (I). Hence, by (III), $g(g(T_1, T_2), T_3) \succ_{\mathrm{rpo}} g(T_1, T_3)$.
3. Similarly.
4. By (I), $g_2(T_1, T_2) \succ_{\mathrm{rpo}} T_1$ and $g_2(T_1, T_2) \succ_{\mathrm{rpo}} T_2$. So by (III), $g_1(g_2(T_1, T_2), T_3) \succ_{\mathrm{rpo}} g_1(T_1, T_3)$ and $g_1(g_2(T_1, T_2), T_3) \succ_{\mathrm{rpo}} g_1(T_2, T_3)$. Hence by (II),
   as $g_1 \succ g_2$, $g_1(g_2(T_1, T_2), T_3) \succ_{\mathrm{rpo}} g_2(g_1(T_1, T_3), g_1(T_2, T_3))$.
5. Similarly.
6. $g(T_1, T_2) \succ_{\mathrm{rpo}} T_1$ by (I).
7. Let $g_1 \equiv y = x$ and $g_2 \equiv x = y$. By the side condition, $g_1 \succ g_2$. By (I), $g_1(T_1, T_2) \succ_{\mathrm{rpo}} T_1$ and $g_1(T_1, T_2) \succ_{\mathrm{rpo}} T_2$, so by (II) $g_1(T_1, T_2) \succ_{\mathrm{rpo}} g_2(T_1, T_2)$.
8. Let $g \equiv x = y$, and assume the proviso, $y \succ x$. By lemma 9, $T_1[y] \succ_{\mathrm{rpo}} T_1[x]$. By monotonicity of $\succ_{\mathrm{rpo}}$, $g(T_1[y], T_2) \succ_{\mathrm{rpo}} g(T_1[x], T_2)$.

This proves the containment of the rewrite relation in $\succ_{\mathrm{rpo}}$, so it is terminating [10]. ⊠

Corollary 11 *Every EQ-BDD is equivalent to some EQ-OBDD.*

2.3 Satisfiability checking

Traditional OBDDs are unique representations of boolean functions, which makes OBDDs very useful to check equivalence between formulae. For EQ-OBDDs, however, this uniqueness property does not hold, as the following examples show.

Example 12 *Let $x \prec y \prec z$. Consider the EQ-BDDs* $\mathrm{ITE}(x = y, \mathbf{1}, \mathrm{ITE}(y = z, \mathbf{0}, \mathbf{1}))$ *and* $\mathrm{ITE}(x = z, \mathbf{1}, \mathrm{ITE}(y = z, \mathbf{0}, \mathbf{1}))$. *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 clearly not identical.*

Another example is formed by the following EQ-BDDs: $\mathrm{ITE}(y = z, \mathbf{1}, \mathbf{0})$ *and* $\mathrm{ITE}(x = y, \mathrm{ITE}(x = z, \mathbf{1}, \mathbf{0}), \mathrm{ITE}(y = z, \mathbf{1}, \mathbf{0}))$. *Here the redundant test $x = y$ is not removed, because in the left-subtree a substitution took place. The situation is depicted in Figure 2.*

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 13 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. We write $\alpha \sqsubseteq \beta$ if α is a prefix of β, i.e. there exists a path γ such that $\alpha.\gamma \equiv \beta$. Let T be an EQ-BDD. We define the set of sequences of T, notation $seq(T)$, as follows:*

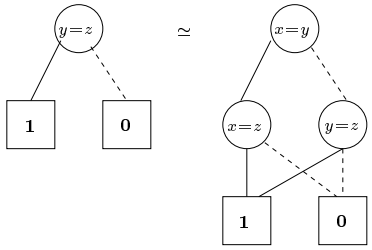


Figure 2: Two logically equivalent EQ-OBDDs

- $seq(\mathbf{0}) = \emptyset$ *and* $seq(\mathbf{1}) = \emptyset$.
- $seq(\mathrm{ITE}(g, T_1, T_2)) = \{\varepsilon\} \cup \{1.\alpha \mid \alpha \in seq(T_1)\} \cup \{0.\alpha \mid \alpha \in seq(T_2)\}$.

For a path $\alpha \in seq(T)$ *we write* $T|_\alpha$ *for the* guard at the end of path α*, inductively defined by:*

- $\mathrm{ITE}(G, T_1, T_2)|_\varepsilon = G$.
- $\mathrm{ITE}(G, T_1, T_2)|_{1.\alpha} = T_1|_\alpha$ *(the high branch).*
- $\mathrm{ITE}(G, T_1, T_2)|_{0.\alpha} = T_2|_\alpha$ *(the low branch).*

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

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

Example 14 *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:* $Th(T, 0.1) = \{x \neq y, y = z\}$.

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 15 *Let* T *be an EQ-OBDD, and* $\alpha, \beta \in seq(T)$ *be 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* $Th(T, \alpha) \vdash x = z$ *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) would be applicable. 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 labelled with $y = z$ and $x = y$, at least one of the steps (4,5) would be applicable. Both cases contradict orderedness of T.

(3) From $Th(T, \alpha) \vdash x = z$, we can show by considering the shortest sequence of equations proving $x = z$ that for some y (possibly $x \equiv y$), either

- $y = z \in Th(T, \alpha)$ and $Th(T, \alpha) \setminus \{y = z\} \vdash x = y$. In this case the lemma obviously holds. Or,

- $z = y \in Th(T, \alpha)$ and $Th(T, \alpha) \setminus \{z = y\} \vdash x = y$. But this is impossible for ordered T, as by (1) and (2), y does not occur in $Th(T, \alpha) \setminus \{z = y\}$. So, $Th(T, \alpha) \setminus \{z = y\} \nvdash x = y$.

$\boxtimes$

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

Lemma 16 *Let T be an EQ-OBDD and $\alpha \in seq(T)$. It holds that*

1. $Th(T, \alpha) \nvdash T|_\alpha$ *and*
2. $Th(T, \alpha) \nvdash \neg T|_\alpha$.

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

(1) Assume $Th(T, \alpha) \vdash x = z$. By Lemma 15.3, for some y, $y = z \in Th(T, \alpha)$. Then rewrite step 8 is applicable, which contradicts orderedness.

(2) Assume $Th(T, \alpha) \vdash x \neq z$. By Lemma 15.2, no positive equations containing z occur in $Th(T, \alpha)$. Hence for some y, $y \neq z \in Th(T, \alpha)$ and $Th(T, \alpha) \vdash y = x$. Then $y \prec x \prec z$ and by Lemma 15.3, for some w, $w = x \in Th(T, \alpha)$. But then rewrite step 8 would be applicable, which contradicts orderedness. $\boxtimes$

Combining the two previous lemmas, we can prove that in an ordered EQ-BDD, each path is consistent.

Theorem 17 *Let T be an EQ-OBDD and $\alpha \in seq(T)$ be a path. Then $Th(T, \alpha)$ is consistent.*

Proof: With induction on α, we can prove that $Th(T, \alpha)$ is consistent:

- $Th(T, \varepsilon) = \emptyset$, which is consistent.
- $Th(T, \alpha.1) = Th(T, \alpha) \cup \{T|_\alpha\}$, which is consistent by the induction hypothesis and lemma 16.1.
- $Th(T, \alpha.0) = Th(T, \alpha) \cup \{\neg T|_\alpha\}$, which is consistent by the induction hypothesis and lemma 16.2.

$\boxtimes$

Corollary 18 *It is now obvious to conclude the following:*

- *The only tautological EQ-OBDD is* $\mathbf{1}$.
- *The only contradictory EQ-OBDD is* $\mathbf{0}$.
- *All other EQ-OBDDs are satisfiable only.*

Proof: As every path in an EQ-OBDD is consistent, there exist a suitable domain D and interpretation functions I and J which allows this path to be taken. So, if the EQ-OBDD is tautological, every path must end in a $\mathbf{1}$. Because rewrite rule 1 of ORDER is not applicable, the EQ-OBDD is equal to $\mathbf{1}$. Similarly, for contradictions. $\boxtimes$

3 Algorithm for checking tautology and satisfiability

The previous sections introduce EQ-BDDs with their properties and give a rewrite system to order them. We are now interested in constructing EQ-BDDs out of formulae. In traditional BDDs, a formula is inductively translated into OBDDs directly. We will call this the "bottom-up" algorithm. Given two ordered BDDs, the logical operations (conjunction, disjunction, etc.) can be performed in polynomial time by Bryant's APPLY algorithm. It is not clear how to generalize this to an efficient method for EQ-OBDDs, however. When two EQ-OBDDs are combined, new substitutions must be done in both of them. After these substitutions, the sub-OBDDs will not be ordered, in general. 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 a solution we use a top-down approach, which in the context of OBDDs has for instance been described in [16]. This approach is based on the Shannon expansion. For propositional logic, this reads:

$$\Phi \iff (\neg p \wedge \Phi|_{\neg p}) \vee (p \wedge \Phi|_{p}) \iff \mathrm{ITE}(p, \Phi|_{p}, \Phi|_{\neg p})$$

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 subfunctions 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:

$$\begin{array}{rcl} \Phi & \iff & (x \neq y \wedge \Phi|_{x\neq y}) \vee (x = y \wedge \Phi|_{x=y}) \\ & \iff & \mathrm{ITE}(x = y, \Phi|_{x=y}, \Phi_{x\neq y}) \end{array}$$

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. But due to the substitutions it is not guaranteed that the resulting EQ-BDD is ordered. However, we show that repeatedly applying the Shannon expansion does lead to an EQ-OBDD.

3.1 A topdown algorithm

We now describe the algorithm precisely and prove soundness and termination. 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 19 *The TRS* SIMPLIFY *consists of the following rules:*

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

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

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. Note that every closed formula rewrites to $\mathbf{0}$ or $\mathbf{1}$. Also note that ordered EQ-BDDs are simplified.

Definition 20 *We define $\Phi|_s$, where s is p, $\neg p$, $x = y$ or $x \neq y$ inductively on the structure of Φ. We start with the case where Φ is a guard g. In this case, $\Phi|_s$ is defined via the following table, where g is put in the rows, and s in the columns:*

	p	$x = y$
q	$\mathbf{1}$ if $p \equiv q$ q if $p \not\equiv q$	q
$u = v$	$u = v$	$\mathbf{1}$ if $u \equiv x$ and $v \equiv y$ $x = v$ if $u \equiv y$ and $v \not\equiv y$ $u = x$ if $u \not\equiv y$ and $u \not\equiv x$ and $v \equiv y$ $u = v$ if $u \not\equiv y$ and $v \not\equiv y$

	$\neg p$	$x \neq y$
q	$\mathbf{0}$ if $p \equiv q$ q if $p \not\equiv q$	q
$u = v$	$u = v$	$\mathbf{0}$ if $u \equiv x$ and $v \equiv y$ $u = v$ otherwise

We now continue the recursive definition of Φ:

$$\begin{aligned}
&\mathbf{0}|_s \equiv \mathbf{0}\\
&\mathbf{1}|_s \equiv \mathbf{1}\\
&(\neg\Phi)|_s \equiv \neg(\Phi|_s)\\
&(\Phi_1 \wedge \Phi_2)|_s \equiv (\Phi_1|_s) \wedge (\Phi_2|_s)\\
&\mathrm{ITE}(\Phi_1, \Phi_2, \Phi_3)|_s \equiv \mathrm{ITE}(\Phi_1|_s, \Phi_2|_s, \Phi_3|_s)
\end{aligned}$$

Example 21 *Let $\Phi \equiv x = z \wedge y = z$ and $g \equiv x = z$ and assume $x \prec y \prec z$. Then $\Phi|_g \equiv \mathbf{1} \wedge y = x$ 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 basis top-down transformation algorithm:

Definition 22 *Let Φ be a simplified formula. We define the algorithm* Topdown *as follows:*

- $\textsc{Topdown}(\mathbf{1}) \equiv \mathbf{1}$
- $\textsc{Topdown}(\mathbf{0}) \equiv \mathbf{0}$
- *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 due to substitions, new equalities can be introduced, but this happens on the fly. Therefore we need no heuristics to limit the quadratic number of possible equations. We have termination and soundness of the algorithm Topdown.

It is clear that for any formula Φ, $\textsc{Topdown}(\Phi)$ always yields a simplified EQ-BDD. We now show that the algorithm Topdown terminates, by using as a size $\#(\Phi)$, the number of occurrences of guards in the EQ-BDD, viewed as tree.

Definition 23

$$\begin{aligned}
\#(\mathbf{0}) &= 0\\
\#(\mathbf{1}) &= 0\\
\#(p) &= 1\\
\#(x = y) &= 1\\
\#(\neg\Phi) &= \#(\Phi)\\
\#(\Phi \wedge \Psi) &= \#(\Phi) + \#(\Psi)\\
\#(\mathrm{ITE}(\Phi, \Psi, \Xi)) &= \#(\Phi) + \#(\Psi) + \#(\Xi)
\end{aligned}$$

Note that none of the rules increases the number of guards, so we have the following:

Lemma 24 *For any formula Φ, we have $\#(\Phi) \geq \#(\Phi\downarrow)$.*

Lemma 25 *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 the table of Definition 20, each guard is replaced by at most one other guard. ⊠

Theorem 26 *The algorithm* TOPDOWN *always terminates.*

Proof: Within TOPDOWN(Φ) there are recursive calls to TOPDOWN($\Phi|_g\downarrow$) and TOPDOWN($\Phi|_{\neg g}\downarrow$). By the previous lemmata, $\#(\Phi) > \#(\Phi|_g\downarrow)$ and similar for $\neg g$. So, in each recursive call of TOPDOWN the size strictly decreases, and hence TOPDOWN must terminate. ⊠

Now we show that the algorithm TOPDOWN is sound in the sense that TOPDOWN(Φ) is equivalent to Φ.

Lemma 27 *For any formula Φ, simplified formula Ψ and simplified guard g, we have:*

$$(1)\quad \Phi \Longleftrightarrow \Phi\downarrow \qquad (2)\quad g \Rightarrow (\Psi \Longleftrightarrow \Psi|_g) \qquad (3)\quad \neg g \Rightarrow (\Psi \Longleftrightarrow \Psi|_{\neg g})$$

Proof: (1) Each rewrite step is sound. (2,3) It is easy to check this for guards. For arbitrary Ψ it then follows by structural formula induction. ⊠

Theorem 28 (soundness) *For any formula Φ, we have:* $\Phi \Longleftrightarrow$ TOPDOWN(Φ)

Proof: Induction over $\#(\Phi)$. The base case is trivial. The induction step is as follows:

	Φ	
$\Longleftrightarrow$	$\mathrm{ITE}(g, \Phi, \Phi)$	Idempotence
$\Longleftrightarrow$	$\mathrm{ITE}(g, \Phi|_g, \Phi|_{\neg g})$	Lemma 27.(2,3)
$\Longleftrightarrow$	$\mathrm{ITE}(g, \Phi|_g\downarrow, \Phi|_{\neg g}\downarrow)$	Lemma 27.1
$\Longleftrightarrow$	$\mathrm{ITE}(g, \textsc{Topdown}(\Phi|_g\downarrow), \textsc{Topdown}(\Phi|_{\neg g}\downarrow))$	Induction hypothesis
$\Longleftrightarrow$	$\overline{\mathrm{ITE}}(g, \textsc{Topdown}(\Phi|_g\downarrow), \textsc{Topdown}(\Phi|_{\neg g}\downarrow))$	Idempotence

⊠

3.2 Iteration of TOPDOWN

Unfortunately, it is *not* the case that TOPDOWN(Φ) is always ordered, for (at least) two different reasons, as the following example shows. These situations are depicted in Figure 3. In this figure, the call graph of TOPDOWN is depicted, which corresponds to the non-reduced EQ-BDD (i.e. lacking the idempotence rule 1 of Definition 6). Here dashed nodes are not in the EQ-BDD. The argument of TOPDOWN is put in square brackets.

Example 29 *Assume $x \prec y \prec z$.*

- *Consider* TOPDOWN$(x = z \wedge y = z) \equiv \mathrm{ITE}(x = z, \mathrm{ITE}(x = y, \mathbf{1}, \mathbf{0}), \mathbf{0})$. *First, the smallest guard in Φ is put at the top and it is used to substitute the high branch. This may create an even smaller guard.*

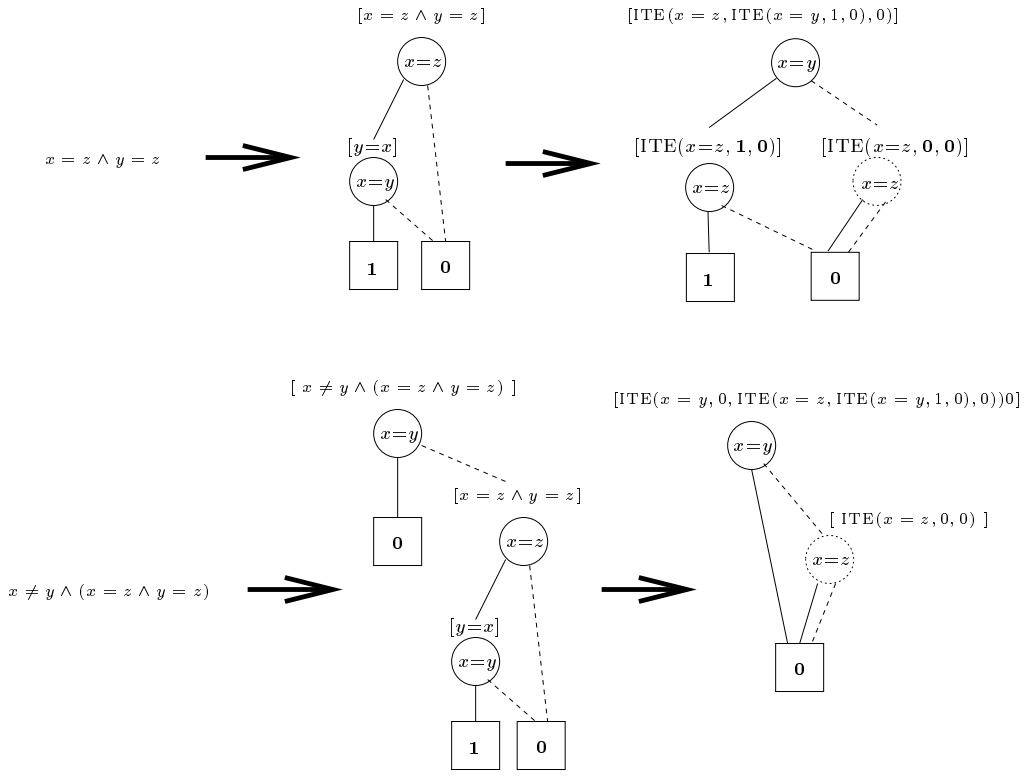


Figure 3: Examples where TOPDOWN does not yield an EQ-OBDD

- *The second example is*

$$\begin{aligned}&\textsc{Topdown}(x \neq y \wedge (x = z \wedge y = z)) \equiv \\ &\qquad \mathrm{ITE}(x = y, \mathbf{0}, \mathrm{ITE}(x = z, \mathrm{ITE}(x = y, \mathbf{1}, \mathbf{0}), \mathbf{0}))\mathbf{0}.\end{aligned}$$

  In the low branch, $x = y$ is replaced by $\mathbf{0}$, but due to substitutions in the recursive call, new occurrences of $x = y$ may be generated. Note that this second example is a dangerous one 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.*

The following lemmas and subsequent corollary indicate how an EQ-OBDD can be constructed. 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 2 iterations of Topdown were required to obtain an EQ-OBDD. We now prove that the fixed point will always be reached, and that it is an ordered EQ-BDD.

Lemma 30 *Let Φ be a simplified EQ-BDD and g be a simplified guard. It holds that:*

(1) $\Phi \succeq_{\mathrm{rpo}} \Phi|_g\downarrow$ (3) *if g occurs in Φ, then* $\Phi \succ_{\mathrm{rpo}} \Phi|_g\downarrow$

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

Proof: We apply recursive path ordering, where $\mathbf{0}$ and $\mathbf{1}$ are lowest in the ordering of function symbols. Moreover, we view each application $\mathrm{ITE}(g, T, U)$ as an application of guard g to arguments T and U, similarly as in the proof of Lemma 10, ordered by $\succ$ of Definition 5.

Proofs can be given with simultaneous induction on the structure of Φ. We provide two interesting fragments of the proof, namely where Φ is of the form $\mathrm{ITE}(u = v, T, U)$ and g has the form $x = y$ in the cases (1) and (3). Note that $x \prec y$ and $u \prec v$, because Φ and g are simplified.

First consider case (1). By definition $\Phi|_g\downarrow \equiv \mathrm{ITE}((u = v)|_g\downarrow, T|_g\downarrow, U|_g\downarrow)\downarrow$. First 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, $T \succeq_{\mathrm{rpo}} T|_g\downarrow$. By property (I) of recursive path orderings it follows that $\Phi \succ_{\mathrm{rpo}} T$ and hence $\Phi \succ_{\mathrm{rpo}} \Phi|_g\downarrow$. 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 $\Phi \succ_{\mathrm{rpo}} T|_g\downarrow$ and $\Phi \succ_{\mathrm{rpo}} U|_g\downarrow$. So, by property (II) of RPOs it follows that $\Phi \succ_{\mathrm{rpo}} \Phi|_g\downarrow$. In the last case, where $(u = v)|_g\downarrow \equiv u = v$, we find by the induction hypothesis $T \succeq_{\mathrm{rpo}} T|_g\downarrow$ and $U \succeq_{\mathrm{rpo}} U|_g\downarrow$. By property (III) of RPOs it follows that $\Phi \succeq_{\mathrm{rpo}} \Phi|_g\downarrow$.

Now consider case (3). Note that in case (1) we proved that $\Phi \succ_{\mathrm{rpo}} \Phi|_g\downarrow$ 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 $T \succ_{\mathrm{rpo}} T|_g\downarrow$. Furthermore, by case (1) $U \succeq_{\mathrm{rpo}} U|_g\downarrow$. So, by property (III) of RPOs we can conclude that

$$\begin{aligned}&\Phi \equiv \\ &\mathrm{ITE}(u = v, T, U) \succ_{\mathrm{rpo}} \\ &\mathrm{ITE}(u = v, T|_g\downarrow, U|_g\downarrow) \equiv \\ &\mathrm{ITE}((u = v)|_g\downarrow, T|_g\downarrow, U|_g\downarrow) \equiv \\ &\Phi|_g\downarrow.\end{aligned}$$

$\boxtimes$

Lemma 31 *Let Φ be a simplified EQ-BDD.*

1. $\Phi \succeq_{\mathrm{rpo}} \textsc{Topdown}(\Phi)$.

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

Proof: The proof of case 1 is given first. We prove this theorem with induction on $\#(\Phi)$. Note that if Φ does not contain a guard then it is equal to $\mathbf{1}$ or $\mathbf{0}$, 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 24, 25 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 30, 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 $\textsc{Topdown}(\Phi|_g\downarrow) \equiv \texttt{topdown}(\Phi|_{\neg g}\downarrow)$. Then $\textsc{Topdown}(\Phi) \equiv \textsc{Topdown}(\Phi|_g\downarrow)$ and we are done by (*).

Now assume $\textsc{Topdown}(\Phi|_g\downarrow) \not\equiv \texttt{topdown}(\Phi|_{\neg g}\downarrow)$. Then

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

Note that Φ must have the form $\Phi \equiv \text{ITE}(h, T, U)$. 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 30 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 we can conclude that $\Phi \succ_{\text{rpo}} \textsc{Topdown}(\Phi)$.

Now we provide the proof of the second item of this lemma. This proof is split in the following cases:

$\Longrightarrow$ We must show that if Φ is ordered, then $\Phi \equiv \texttt{topdown}(\Phi)$. We prove this by induction on the structure of Φ. The case where Φ equals $\mathbf{0}$ or $\mathbf{1}$ is trivial. So, consider the case where $\Phi \equiv \text{ITE}(g, T, U)$. As Φ is ordered and g is the smallest guard of Φ, g does not occur in T or U. Also, if $g \equiv x = y$, y does not occur in T. Moreover, T and U are ordered, hence it is also simplified. So, $\Phi|_g\downarrow \equiv T$ and $\Phi|_{\neg g}\downarrow \equiv U$. Note that $T \not\equiv U$.

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

$\Longleftarrow$ In this case we must show that if $\Phi \equiv \textsc{Topdown}(\Phi)$, then Φ is ordered. So, suppose Φ is not ordered. Then one of the rules of Order (Definition 6) can be applied. It is clear by the construction of $\textsc{Topdown}(\Phi)$, that rules 1, 6, 7 and 8 are not applicable, so on some part the guards are not strictly increasing. Locate the largest sub-EQ-BDD Ψ in Φ of the form $\text{ITE}(g, T, U)$ where g is not bigger than all guards below it in Ψ. Obviously, $\Psi \not\equiv \textsc{Topdown}(\Psi)$. Now a simple inductive argument shows that $\Phi \not\equiv \textsc{Topdown}(\Phi)$.

$\boxtimes$

Corollary 32 *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 31.1 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 31.2 Φ_i is the required EQ-OBDD. Note that by Lemma 31.2 Φ_i is the first ordered EQ-BDD in the sequence. $\boxtimes$

We conclude with the complete algorithm to transform an arbitrary formula Φ to EQ-OBDD:

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

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

Claim 33 *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 check the benchmarks reported in [17, 19]. They report to have comparable performance as in [11]. Unfortunately, we could not obtain the benchmarks used in [8]. We first describe the benchmarks, then the implementation, including some variable orderings we used, and finally present the results.

3.3.1 Benchmarks

The benchmark formulae can be obtained from [19] and most of them could be solved with the methods described in [17]. 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 [17] 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:

- The domain variables range over the non-negative integers, and integer constants are allowed.
- Certain variables are declared as boolean variables and range over the subset $\{\mathbf{0}, \mathbf{1}\}$ of the integers. Boolean variables can occur in equations.
- A special constant -1 is present, which is not an element of the domain of the variables.
- The *ITE* construct is applied on arbitrary integer expressions.
- The formulae are stored in shared form, by using macro-definitions. For complexity-considerations they should be regarded as circuits.

Example 34 *Let p be a boolean variable, and let x and y be integers. The following is a typical example of a formula in this extended format: Let $X \equiv p = \text{ITE}(x = y, \mathbf{0}, 2)$. If $x = y$, this evaluates to $p = \mathbf{0}$, i.e. $\neg p$; otherwise, it reduces to $p = 2$, i.e. $\mathbf{0}$ (false), as boolean variables range over $\{\mathbf{0}, \mathbf{1}\}$.*

Note that in $5 = \text{ITE}(p = x, 5, 4)$, $p = x$ should not *be regarded as a guard. The only guard in this example is p. This formula is equivalent to $(p \wedge x = 1) \vee (\neg p \wedge x = 0)$.*

These extensions can be dealt with by adding some new reduction rules:

$$
\begin{array}{rcll}
m = n & \to & \mathbf{0} & \text{if } m \text{ and } n \text{ are different integer constants} \\
x = -1 & \to & \mathbf{0} & \text{if } x \text{ is a domain variable} \\
-1 = x & \to & \mathbf{0} & \text{if } x \text{ is a domain variable} \\
p = m & \to & \mathbf{0} & \text{if } p \text{ is a boolean variable,} \\
 & & & \text{and } m \text{ is an integer constant different from } \mathbf{0} \text{ and } \mathbf{1} \\
m = p & \to & \mathbf{0} & \text{if } p \text{ is a boolean variable,} \\
 & & & \text{and } m \text{ is an integer constant different from } \mathbf{0} \text{ and } \mathbf{1}
\end{array}
$$

We now have a new type of guards, of the form $c = x$, where c is an integer constant. The ordering on guards is extended as follows, where each entry displays the condition when the guards in the left are smaller than the guards on the right. This is a total ordering on oriented guards, provided the sets of boolean and integer variables are disjoint. In the following, p and q are boolean variables, u and v are integer variables and c and d are integer constants.

$\prec$	q	$u = v$	$d = u$
p	$p \prec q$	$p \prec u$	$p \prec u$
$x = y$	$x \prec q$	$x \prec u \vee$ $(x \equiv u \wedge y \prec v)$	$x \prec u$
$c = x$	$x \prec q$	$x \preceq u$	$x \prec u \vee$ $(x \equiv u \wedge c < d)$

3.3.2 Prototype implementation

We have made a prototype implementation of the TOPDOWN algorithm. As programming language we used C, including the `ATerm`-library [5]. 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 $(\Psi|_p\downarrow)|_{\neg q}\downarrow \equiv (\Psi|_{\neg p}\downarrow)|_q\downarrow$. Due 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. refraining from equations) it builds an OBDD from a propositional formula in one iteration. Due to the various hash tables, the memory demands are rather high. This memory consumption can easily be optimized, for instance by using result pointers instead of hash tables for memoization, as in [2].

So far we have considered arbitrary variable orderings, but an implementation must choose one. It is well known that the size of an OBDD and the efficiency of the BDD-operations crucially depend on the variable ordering. It is NP-hard to find the optimal variable ordering. Various heuristics exist for guessing a good ordering on the variables. We implemented two of these heuristics. Both heuristics are taken over from the propositional case; see [16] for a detailed description.

Nr. file	[17, 19]	t	bt	wft	bwft	bft	br	r
022	:0.16	:31	:51	:39	3:59	—	28:25	31:04
025	:0.2	:1.1	:1.2	:1.1	:7.3	:8.0	:0.5	:0.5
027	:1.7	16:37	32:18	16:30	—	—	—	—
032	:0.1	:7.9	:8.7	:8.4	14:0	38:42	14:19	15:47
037	:0.15	8:27	:11	8:28	:12	:18	:49.2	5:43
038	:0.18	1:22	:1.7	1:22	:1.8	:2.2	:1.8	:1.5
043	—	—	—	—	—	—	—	—
044	:0.1	:4.4	:2.1	:3.4	:4.0	:3.2	:6.1	1:41
046	:0.13	—	—	—	—	—	—	—
049	—	—	—	—	28:47	—	:0.2	:0.3

Figure 4: Timing results for the benchmarks

Both heuristics construct an ordering by inspecting the original formula. Consider the formula, represented as a maximally shared DAG. Note that due to maximal sharing, each variable occurs at most once. The `fanin` heuristic chooses a variable with the maximal number of incoming edges as the smallest variable; this is a local property of the DAG. The `weight` heuristic also takes into account the distance to the root. The motivation is that nodes higher up in the DAG contribute more to the final result. The root gets weight 1, and the other nodes sum up the weights they get from their parents. Furthermore, each node divides its own weight equally among its children. So

$$weight(m) = \begin{cases} 1, \text{ if } m \text{ is the root} \\ \sum_{n \in parents(m)} \frac{weight(n)}{fanout(n)} \end{cases}$$

We computed the weights after a first simplification of the formula, using the term rewriting system of Definition 19. This eliminates some variables that do not contribute to the final result, giving them weight 0. We have used the following basic variable orders:

mnemonic	$a < b$
(t)extual	a is declared earlier than b in the text file
(r)everse	a is declared later than b in the text file
(f)anin	$fanin(a) > fanin(b)$
(w)eights	$weight(a) > weight(b)$
(b)ooleans	a is a boolean variable and b a domain variable

Note that only (t) and (r) are total, so we used lexicographic combinations. The total order $lex(b, f, t)$ for instance means: First use ordering (b); if the results are equal, then use ordering (f); if the results are still equal, use ordering (t).

3.3.3 Results

Having explained the benchmarks and all ingredients of the implementation, we can now present the results. They can be found in Figure 4. The first column contains the number of the files, as given in [19]. The second column contains the times reported in [19], obtained by the method of [17]. The other columns show our results, using various variable orderings, as explained earlier (here bft means $lex(b, f, t)$). 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 250 MHz and where the processes could use up to 1.5 GB internal memory.

Nr. file	t	bt	wft	bwft	bft	br	r
022	1	1	1	1	—	1	1
025	1	1	1	1	1	1	1
027	2	2	2	—	—	—	—
032	2	2	2	1	2	1	1
037	1	1	1	1	1	1	1
038	1	1	1	1	1	1	1
043	—	—	—	—	—	—	—
044	2	2	2	2	2	2	2
046	—	—	—	—	—	—	—
049	—	—	—	1	—	1	1

Figure 5: Number of iterations for the benchmarks

The table shows that we can solve 8 out of 10 formulae. In this respect our method is comparable to [17]. 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. [1]

It is also clear that the variable ordering is very important, and that the textual ordering, as provided by [19] happens to coincide with the weighted version quite well.

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. The results can be found in Figure 5

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 example, it is more effective than using APPLY, especially in combination with the `weight` heuristics.

Example 35 *Consider the formula $X \equiv p \wedge (\Phi \wedge \neg p)$. Note that weight$(p) \geq 3/4$. Hence p will be the smallest variable. Note that $X|_p\downarrow \equiv \mathbf{0}$ and $X|_{\neg p}\downarrow \equiv \mathbf{0}$, so* TOPDOWN *terminates in one call, detecting the contradiction. The usual* APPLY *algorithm will completely build the tree for Φ, potentially resulting in an exponential blow-up.*

4 Conclusion

We incorporated equations in BDDs directly, without encodings. The resulting objects are called EQ-BDDs. A straightforward notion of ordered EQ-BDDs is defined, and it is proved that each EQ-BDD is logically equivalent to some EQ-OBDD. Moreover, on EQ-OBDDs satisfiability and tautology checking can be done in constant time.

We also introduced an algorithm, which for any propositional formula with equations finds an EQ-OBDD that is logically equivalent to it. The algorithm is proved to be sound and terminating, by means of recursive path ordering. The algorithm has been implemented, and applied to benchmarks known from literature. The performance is comparable to existing methods.

Future work. Various improvements within our framewrok are still possible. To mention only a few, we could add other rewrite rules, in order to recognize structural properties of the formulae,

[1] We plan to do such optimizations before the final paper is delivered.

like $\Phi \wedge \neg\Phi \Rightarrow \mathbf{0}$. This technique is also used in [2]. The BEDs introduced in that paper are quite similar to the objects in our implementation: maximally shared term representations with nodes for guards and boolean connectives. Other improvements would be to investigate whether various other BDD-techniques can be incorporated, like choosing (and changing) the variable ordering dynamically, complemented edges, ITE-standard triples etc. In order to reduce the needed iterations of TOPDOWN, it could be checked before the recursive call to TOPDOWN whether a smaller equation occurs in $\Phi|_g \downarrow$. We have not investigated the effect, because the benchmarks terminated after at most 2 iterations.

We think that it is important future work to extend EQ-BDDs with function symbols. Our original motivation behind this article comes from our investigation into the analysis of distributed systems and protocols. The behaviour of these systems is described in the process algebra μCRL [12], in the style of LOTOS [4]. By applying the ideas of for instance [13], properties of the state space are expressed as huge formulae, mainly consisting of the general boolean connectives, the ITE predicate and equations between arbitrary data terms. It quickly becomes obvious that we need automatic means to at least reduce the size of these formulae, and hence we started investigating EQ-BDDs.

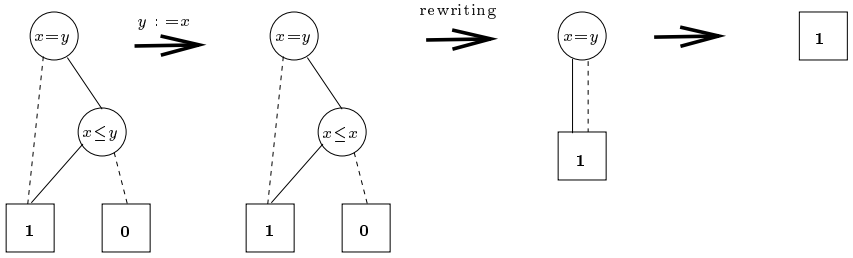
In distributed system specifications, data is usually specified by algebraic data types. Reasoning on data is usually based on term rewriting. For this reason, function symbols cannot be eliminated, and the domains are generally structured and often infinite. Also interactive theorem provers like PVS [18] would greatly benefit from BDD-procedures that deal with equations and function symbols adequately.

Contrary to the existing proposals [11, 8, 17], our approach forms an extendible basis. We might allow function symbols in EQ-BDDs. In the algorithm, the rewrite rules of the data domain can be applied to the TRS SIMPLIFY.

In this way, one is able to prove that $x \leq y \vee x \neq y$ is a tautology. Obviously this is not true when the interpretation of functions is free (eg. interpret $\leq$ as $<$). However, consider the following definition of $\leq$ in terms of rewrite rules, where S denotes the succesor function:

$$\begin{array}{lcl} x < 0 & \rightarrow & \mathbf{0} \\ x < S(y) & \rightarrow & x \leq y \\ x \leq y & \rightarrow & x < y \vee x = y \end{array}$$

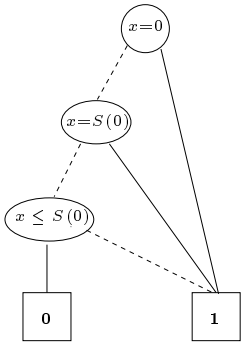
An EQ-BDD proof with auxiliary rewrite rules of $x \leq y \vee x \neq y$ looks as follows:



Note that combining EQ-BDDs with additional rewrite rules is not fully trivial. For instance, a term like $x \leq y$ cannot be further reduced. It would be interesting to see under which conditions the basic theory on the existence of ordered EQ-BDDs and the algorithm can be extended to function symbols.

Another point requiring attention is that in general, algebraic data domains force infinite domains, disallowing the use of the finite domain property. Consider for instance $\forall y.\, x \geq y$ where x and y are natural numbers. This formula is obviously false. However, in a finite interpretation of the natural numbers this formula would be satisfiable.

Sometimes, a formula can only be proven using the structure of a data domain. Consider for instance $x \leq S(0) \rightarrow x = 0 \vee x = S(0)$, which yields the following EQ-OBDD



In order to show that this OBDD a tautology, it is required to know that each natural number can either be written as 0 or as $S(y)$ for some natural number y. This requires the use of case distinction and induction principles, a hard field which has as far as we know never been addressed in the realm of BDDs.

Acknowledgments. We like to thank Ofer Shtrichman for making his benchmarks publicly available, and for his kind help in explaining their syntax and semantics.

References

[1] Wilhelm 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] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[5] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software Practice en Experience*, (accepted). Paper and software accessible via `http://www.wins.uva.nl/~olivierp/aterm.html`.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[7] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[8] Randal E. Bryant, Steven German, and Miroslav N. Velev. Exploiting prositive equality in a logic of equality with uninterpreted functions. In *CAV'99*, 1999.

[9] Randal E. Bryant, Steven German, and Miroslav N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. Technical Report CMU-CS-99-115, Carnegie Mellon University, Pittsburgh, May 1999.

[10] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1–2):69–115, 1987.

[11] A. Goel, K. Sajid, H. Zhou, and A. Aziz. BDD based procedures for a theory of equality with uninterpreted functions. In *CAV'98*, number 1427 in LNCS, pages 244–255, 1998.

[12] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes, Utrecht 1994*, Workshops in Computing, pages 26–62. Springer, 1994.

[13] J.F. Groote and M.P.A. Sellink. Confluence for process verification. *Theoretical Computer Science B (Logic, semantics and theory of programming)*, 170(1–2):47–81, 1996.

[14] J.R. Burch and D.L. Dill. Automatic verification of pipelined microprocessors control. In D. L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer-Verlag, June 1994.

[15] 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.

[16] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design: OBDD-Foundations and Applications.* Springer-Verlag, 1998.

[17] Amir Pnueli, Yoav Rodeh, Ofer Shtrichman, and Michael Siegel. Deciding equality formulas by small domains instantiations. In *CAV'99*, 1999.

[18] 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 the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer Verlag, 1996.

[19] O. Shtrichman. Benchmarks for satisfiability checking of equality formulas. See `http://www.wisdom.weizmann.ac.il/~ofers/sat/bench.htm`, 1999.