Knowledge Compilation for Model Counting: Affine Decision Trees*

Frédéric Koriche[1], **Jean-Marie Lagniez**[2], **Pierre Marquis**[1], **Samuel Thomas**[1]
[1]CRIL-CNRS, Université d'Artois, Lens, France
[2]FMV, Johannes Kepler University, Linz, Austria
{koriche,marquis,thomas}@cril.fr Jean-Marie.Lagniez@jku.at

Abstract

Counting the models of a propositional formula is a key issue for a number of AI problems, but few propositional languages offer the possibility to count models efficiently. In order to fill the gap, we introduce the language EADT of *(extended) affine decision trees*. An extended affine decision tree simply is a tree with affine decision nodes and some specific decomposable conjunction or disjunction nodes. Unlike standard decision trees, the decision nodes of an EADT formula are not labeled by variables but by affine clauses. We study EADT, and several subsets of it along the lines of the knowledge compilation map. We also describe a CNF-to-EADT compiler and present some experimental results. Those results show that the EADT compilation-based approach is competitive with (and in some cases is able to outperform) the model counter Cachet and the d-DNNF compilation-based approach to model counting.

1 Introduction

Model counting is a key issue in a number of AI problems, including inference in Bayesian networks and contingency planning [Littman *et al.*, 2001; Bacchus *et al.*, 2003; Sang *et al.*, 2005; Darwiche, 2009]. However, this problem is computationally hard (#P-complete) [Valiant, 1979]. Accordingly, few propositional languages offer the possibility to count models *exactly* in an efficient way [Roth, 1996].

The knowledge compilation (KC) map, introduced by Darwiche and Marquis [2002] and enriched by several authors (see among others [Wachter and Haenni, 2006; Subbarayan *et al.*, 2007; Mateescu *et al.*, 2008; Fargier and Marquis, 2008; Darwiche, 2011; Marquis, 2011; Bordeaux *et al.*, 2012]) is a multi criteria evaluation of languages, where languages are compared according to the queries and the transformations they support in polynomial time, as well as their relative succinctness (i.e., their ability to represent information using little space).

*This work is partially supported by FWF, NFN Grant S11408-N23 (RiSE), and by the project BR4CP ANR-11-BS02-008 of the French National Agency for Research.

Among the languages which have been studied and classified according to the KC map, only the language d-DNNF of formulae in deterministic decomposable negation normal form [Darwiche, 2001], together with is subsets $\text{OBDD}_<$ [Bryant, 1986], FBDD [Gergov and Meinel, 1994], and SDD [Darwiche, 2011], satisfy the **CT** query (model counting). Yet, another interesting language which satisfies **CT** is AFF, the set of all affine formulae [Schaefer, 1978], defined as finite conjunctions of affine clauses (aka XOR-clauses). Unfortunately, AFF is not a *complete* language, because some propositional formulae (e.g. the clause $x \vee y$) cannot be represented into conjunctions of affine clauses.

By coupling ideas from affine formulae and decision trees, this paper introduces a new family of propositional languages that are complete and satisfy **CT**. The blueprint of our family is the class EADT of *extended affine decision trees*. In essence, an extended affine decision tree is a tree with decision nodes and some specific decomposable conjunction or disjunction nodes. Unlike usual decision trees, the decision nodes in an EADT are labeled by affine clauses instead of variables. Our family covers several subsets of EADT, including ADT (the set of affine decision trees where conjunction or disjunction nodes are prohibited), EDT (the set of extended decision trees where decomposable conjunction or disjunction nodes are allowed but decision nodes are mainly restricted to standard ones), and DT, the intersection of ADT and EDT.

Following the lines of the KC map, we prove that ADT and its subclass DT satisfy all queries and transformations offered by ordered binary decision diagrams ($\text{OBDD}_<$). Analogously, EADT and its subclass EDT satisfy all queries offered by d-DNNF and more transformations ($\neg$**C** is not satisfied by d-DNNF). Importantly, we also show that none of $\text{OBDD}_<$, CNF, and DNF is at least as succinct as any of ADT or EADT, and that EADT is strictly more succinct than ADT.

Finally, we describe a CNF-to-EADT compiler (which can be downsized to a compiler targeting ADT, EDT or DT). We used this program to compile a number of benchmarks from different domains. This empirical evaluation aimed at addressing two practical issues: (1) how challenging is an EADT compilation-based approach to model counting, compared to a direct, uncompiled method using a state-of-the-art model counter? (2) how does the EADT compilation-based approach perform compared to a d-DNNF compilation-based method? Our experimental results show that the EADT compilation-based approach is competitive with both methods and, in some cases, it really outperforms them.

The rest of the paper is organized as follows. After introducing some background in Section 2, EADT and its subsets are defined and examined along the lines of the KC map in Section 3. In Section 4, our compiler is described and, in Section 5, some empirical results are presented and discussed. Finally, Section 6 concludes the paper. The run-time code of our compiler can be downloaded at http://www.cril.fr/ADT/

2 Preliminaries

We assume the reader familiar with propositional logic (including the notions of model, consistency, validity, entailment, and equivalence). All languages examined in this study are defined over a finite set PS of Boolean variables, and the constants $\top$ (true) and $\bot$ (false).

Affine Formulae. Let PS be a denumerable set of propositional variables. A literal (over PS) is an element $x \in PS$ (a positive literal) or a negated one $\neg x$ (a negative literal), or a Boolean constant $\top$ (true) or $\bot$ (false). An *affine clause* (aka XOR-clause) δ is a finite XOR-disjunction of literals (the XOR connective is denoted by $\oplus$). $var(\delta)$ is the set of variables occurring in δ. δ is *unary* when it contains precisely one literal. Obviously enough, each affine clause can be rewritten in linear time as a *simplified* affine clause, i.e., a finite XOR-disjunction of positive literals occurring once in the formula, plus possibly one occurrence of $\top$ (just take advantage of the fact that $\oplus$ is associative and commutative, and of the equivalences $\neg x \equiv x \oplus \top$, $x \oplus x \equiv \bot$, $x \oplus \bot \equiv x$, viewed as rewrite rules, left-to-right oriented). For instance, the affine clause $\neg x \oplus x \oplus \neg y \oplus \neg z$ can be turned in linear time into the equivalent simplified affine clause $y \oplus z \oplus \top$. An *affine formula* is a finite conjunction of affine clauses.

Knowledge Compilation. For space reasons, we assume the reader has a basic familiarity with the languages CNF, DNF, $\text{OBDD}_<$, SDD, BDD, FBDD, d-DNNF_T, d-DNNF, and DAG-NNF, which are considered in the following (see [Darwiche and Marquis, 2002; Pipatsrisawat and Darwiche, 2008; Darwiche, 2011] for formal definitions). The basic queries considered in the KC map include tests for consistency **CO**, validity **VA**, implicates (clausal entailment) **CE**, implicants **IM**, equivalence **EQ**, sentential entailment **SE**, model counting **CT**, and model enumeration **ME**. The basic transformations are conditioning (**CD**), (possibly bounded) closures under the connectives ($\wedge$**C**, $\wedge$**BC**, $\vee$**C**, $\vee$**BC**, $\neg$**C**), and forgetting (**FO**, **SFO**).

Finally, let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two propositional languages.

- $\mathcal{L}_1$ is *at least as succinct as* $\mathcal{L}_2$, denoted $\mathcal{L}_1 \leq_s \mathcal{L}_2$, iff there exists a polynomial p such that for every formula $\phi \in \mathcal{L}_2$, there exists an equivalent formula $\psi \in \mathcal{L}_1$ where $|\psi| \leq p(|\phi|)$.
- $\mathcal{L}_1$ is *polynomially translatable* into $\mathcal{L}_2$, noted $\mathcal{L}_1 \geq_p \mathcal{L}_2$, iff there exists a polynomial-time algorithm f such that for every $\phi \in \mathcal{L}_1$, $f(\phi) \in \mathcal{L}_2$ and $f(\phi) \equiv \phi$.

$<_s$ is the asymmetric part of $\leq_s$, i.e., $\mathcal{L}_1 <_s \mathcal{L}_2$ iff $\mathcal{L}_1 \leq_s \mathcal{L}_2$ and $\mathcal{L}_2 \not\leq_s \mathcal{L}_1$. When $\mathcal{L}_1 \geq_p \mathcal{L}_2$ holds, every query which is supported in polynomial time in $\mathcal{L}_2$ also is supported in polynomial time in $\mathcal{L}_1$; conversely, every query which is not supported in polynomial time in $\mathcal{L}_1$ unless P = NP is not supported in polynomial time in $\mathcal{L}_2$, unless P = NP.

3 The Affine Family

All propositional languages in our family are subsets of the very general language of *affine decision networks*:

Definition 1 ADN *is the set of all* affine decision networks*, defined as single-rooted finite DAGs where leaves are labeled by a Boolean constant ($\top$ or $\bot$), and internal nodes are $\wedge$ nodes or $\vee$ nodes (with arbitrarily many children) or* affine decision nodes*, i.e., binary nodes of the form $N = \langle \delta, N_-, N_+ \rangle$ where δ is the affine clause labeling N and N_- (resp. N_+) is the left (resp. right) child of N.*

The size $|\Delta|$ of an ADN formula Δ is the sum of number of arcs in it, plus the cumulated size of the affine clauses used as labels in it. For every node N in an ADN formula Δ, $Var(N)$ is defined inductively as follows:

- if N is a leaf node, then $Var(N) = \varnothing$;
- if N is an affine decision node $N = \langle \delta, N_-, N_+ \rangle$, then $Var(N) = var(\delta) \cup Var(N_-) \cup Var(N_+)$;
- if N is a $\wedge$ node (resp. $\vee$ node) with children $N_1, \ldots, N_k$, then $Var(N) = \bigcup_{i=1}^{k} Var(N_i)$.

Clearly, $Var(\Delta) = Var(R_\Delta)$ (where R_Δ is the root of Δ) can be computed in time linear in the size of Δ. Every ADN formula Δ is interpreted as a propositional formula $I(\Delta)$ over $Var(\Delta)$, where $I(\Delta) = I(R_\Delta)$ is defined inductively as:

- if N is a leaf node labeled by $\top$ (resp. $\bot$), then $I(N) = \top$ (resp. $\bot$);
- if N is an affine decision node $N = \langle \delta, N_-, N_+ \rangle$, then $I(N) = ((\delta \oplus \top) \wedge I(N_-)) \vee (\delta \wedge I(N_+))$;
- if N is a $\wedge$ node (resp. $\vee$ node) with children $N_1, \ldots, N_k$, then $I(N) = \bigwedge_{i=1}^{k} I(N_i)$ (resp. $\bigvee_{i=1}^{k} I(N_i)$).

Finally, $\|\Delta\|$ represents the number of models of the ADN formula Δ over $Var(\Delta)$.

The DAG-NNF language considered in [Darwiche and Marquis, 2002] is polynomially translatable into a subset of ADN, where decision nodes have leaf nodes as children. Indeed, every leaf node labeled by a positive literal x (resp. negative literal $\neg x$) in a DAG-NNF formula is equivalent to the affine decision node N labeled by $\delta = x$ and such that $N_- = \bot$ (resp. $= \top$) and $N_+ = \top$ (resp. $= \bot$). Thus, ADN is a highly succinct yet intractable representation language; especially, it does not satisfy the **CT** query unless P = NP. To this point, we need to focus on tractable subsets of ADN.

Let us start with the EADT language, a class of tree-structured formulae defined in term of affine decomposability. Formally, a $\wedge$ (resp. $\vee$) node N with children $N_1, \ldots, N_k$ in an ADN Δ is said to be *affine decomposable* if and only if:

(1) for any $i, j \in 1, \ldots, k$, if $i \neq j$, then $Var(N_i) \cap Var(N_j) = \varnothing$, and

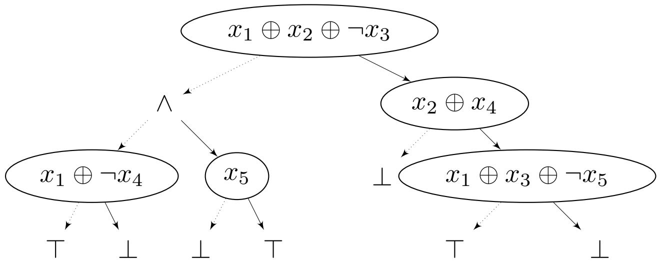

Figure 1: An EADT formula. Every dotted (resp. plain) arc links its source N to N_- (resp. N_+). The formula rooted at the node labeled by $x_2 \oplus x_4$ is an ADT formula.

(2) for every affine decision node N' of Δ which is a parent node of N and which is labelled by the affine clause $\delta_{N'}$, at most one child N_i of N is such that $var(\delta_{N'}) \cap Var(N_i) \neq \varnothing$.

If only the first condition holds, then the node N is said to be *(classically) decomposable*.

Definition 2 EADT *is the set of all* extended affine decision trees*, defined as finite trees where leaves are labeled by a Boolean constant ($\top$ or $\bot$), and internal nodes are affine decision nodes, or affine decomposable $\wedge$ nodes, or affine decomposable $\vee$ nodes.*

An example of EADT formula is given at Figure 1. Some relevant subclasses of EADT are defined as follows:

Definition 3

- ADT *is the set of all* affine decision trees*, i.e., the subset of* EADT *consisting of finite trees where leaves are labeled by a Boolean constant ($\top$ or $\bot$), and internal nodes are affine decision nodes.*
- EDT *is the set of all* extended decision trees*, i.e., the subset of* EADT *where affine decision nodes are labeled by unary affine clauses.*[1]
- DT*, the set of all* decision trees*, is the intersection of* ADT *and* EDT.

Based on this family, it is easy to check that the language TE of all terms and the language CL of all clauses are linearly translatable into DT, and hence, into each of ADT, EDT, and EADT. Furthermore, the language AFF is also polynomially translatable into ADT (hence into its superset EADT).

In contrast to TE, CL, and AFF, the class DT and its supersets ADT, EDT, and EADT are *complete* propositional languages. The completeness property also holds for $\texttt{ODT}_<$, which is the subset of DT consisting of formulae Δ in which every path from the root of Δ to a leaf respects the given, total, strict ordering $<$ (i.e., the variables labeling the decision nodes in the path are ordered in a way which is compatible with $<$). Clearly, $\texttt{ODT}_<$ also is a subset of $\texttt{OBDD}_<$ (to be more precise, $\texttt{ODT}_<$ is the intersection of DT and $\texttt{OBDD}_<$), and both CL and TE are polynomially translatable to it.

We are now in position to explain how any EADT formula Δ can be translated in linear time into a tree $T(\Delta)$ where internal nodes are decomposable $\wedge$ nodes or decomposable $\vee$ nodes or deterministic binary $\vee$ nodes, and the leaves are labeled with affine formulae. The translation T consists in rewriting Δ by parsing it in a top-down way, and collecting sets of affine clauses (those sets are the values of an inherited attribute a defined for each node N of Δ) along the paths of Δ during the translation. T proceeds recursively as follows starting with $N = R_\Delta$ and $a(R_\Delta) = \varnothing$:

- if $N = \langle \delta, N_-, N_+ \rangle$, then $T(N) = T(N_-) \vee T(N_+)$, $a(N_-) = a(N) \cup \{\delta \oplus \top\}$, and $a(N_+) = a(N) \cup \{\delta\}$;
- if $N = \bigwedge_{i=1}^k N_i$, then $T(N) = \bigwedge_{i=1}^k T(N_i)$, and for every $i \in 1, \ldots, k$, $a(N_i) = \{\delta \in a(N) \mid var(\delta) \cap Var(N_i) \neq \varnothing\}$;
- if $N = \bigvee_{i=1}^k N_i$, then $T(N) = \bigvee_{i=1}^k T(N_i)$, and for every $i \in 1, \ldots, k$, $a(N_i) = \{\delta \in a(N) \mid var(\delta) \cap Var(N_i) \neq \varnothing\}$;
- if $N = \top$, then $T(N) = \bigwedge_{\delta \in a(N)} \delta$;
- if $N = \bot$, then $T(N) = \bot$.

By construction, the translation T consists in replacing every affine decision node by a deterministic binary $\vee$ node, every affine decomposable $\wedge$ (resp. $\vee$) node by a (classically) decomposable $\wedge$ (resp. $\vee$) node. Thus, when Δ is an ADT formula, $T(\Delta)$ simply is a deterministic disjunction of affine formulae, and when Δ is a DT formula, $T(\Delta)$ simply is a deterministic DNF formula.

With this translation in hand, it is easy to show that EADT satisfies **CT**. The proof is by structural induction on $\phi = T(\Delta)$. First, $\|\phi\|$ can be computed in polynomial time when ϕ is an affine formula, since ϕ can be viewed as a finite system of linear equations modulo 2. Indeed, ϕ can be turned in polynomial time into its equivalent *reduced row echelon form* ϕ^r, and when $Var(\phi^r)$ contains n variables and ϕ^r contains k affine clauses, $\|\phi\|$ is equal to 2^{n-k}. This solves the base case. As to the inductive step, it is enough to check that:

- if $\phi = \bigwedge_{i=1}^k \phi_i$ (where $\bigwedge$ is a decomposable $\wedge$ node),
$$\|\phi\| = \prod_{i=1}^k \|\phi_i\|$$
- if $\phi = \bigvee_{i=1}^k \phi_i$ (where $\bigvee$ is a decomposable $\vee$ node),
$$\|\phi\| = 2^{|Var(\phi)|} - \prod_{i=1}^k (2^{|Var(\phi_i)|} - \|\phi_i\|)$$
- if $\phi = \phi_1 \vee \phi_2$ (where $\vee$ is a deterministic $\vee$ node), then
$$\|\phi\| = \|\phi_1\| \times 2^{|Var(\phi_2) \setminus Var(\phi_1)|} + \|\phi_2\| \times 2^{|Var(\phi_1) \setminus Var(\phi_2)|}$$

More generally, our results concerning queries and transformations of the KC map are summarized in Proposition 1. Languages d-DNNF and $\texttt{OBDD}_<$ (which are not subsets of EADT) are reported for the comparison matter.

[1] The affine decomposability condition can be given up for EDT formulae since every EDT formula can be translated in linear time into an equivalent EDT formula which is read-once (i.e., for every path from the root of the tree to a leaf, the list of all variables labeling the decision nodes of the path contains at most one occurrence of each variable).

$\mathcal{L}$	**CO**	**VA**	**CE**	**IM**	**EQ**	**SE**	**CT**	**ME**
EADT	$\surd$	$\surd$	$\surd$	$\surd$	?	$\circ$	$\surd$	$\surd$
EDT	$\surd$	$\surd$	$\surd$	$\surd$	?	$\circ$	$\surd$	$\surd$
ADT	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
DT	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
$\texttt{ODT}_<$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
d-DNNF	$\surd$	$\surd$	$\surd$	$\surd$	?	$\circ$	$\surd$	$\surd$
$\texttt{OBDD}_<$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

Table 1: Queries. $\surd$ means "satisfies" and $\circ$ means "does not satisfy unless P = NP."

$\mathcal{L}$	**CD**	**FO**	**SFO**	$\wedge$**C**	$\wedge$**BC**	$\vee$**C**	$\vee$**BC**	$\neg$**C**
EADT	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\surd$
EDT	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\surd$
ADT	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\surd$
DT	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\surd$
$\texttt{ODT}_<$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\surd$
d-DNNF	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	?
$\texttt{OBDD}_<$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\surd$

Table 2: Transformations. $\surd$ means "satisfies," while $\circ$ means "does not satisfy unless P = NP".

Proposition 1 *The results given in Tables 1 and 2 hold.*

In a nutshell, ADT and its subclass DT are equivalent to $\texttt{OBDD}_<$ with respect to queries and transformations. Similarly, EADT and its subclass EDT are essentially equivalent to d-DNNF with respect to queries and transformations. In particular, EDT, EADT, and d-DNNF do not satisfy **SE** unless P = NP, and it is unknown whether they satisfy **EQ**. It is also unknown whether d-DNNF satisfies $\neg$**C**, but this transformation can be done in linear time for both EADT and EDT.

The inclusion graph of the different languages is given in Figure 2. In light of this inclusion graph and the fact that DT (resp. EDT) does not satisfy more queries or transformations than ADT (resp. EADT), it follows that DT (resp. EDT) cannot prove a better choice than ADT (resp. EADT) from the KC point of view. This is why we focus on ADT and EADT in the following. For these languages, we obtained the following succinctness results:

Proposition 2 CNF $\not\leq_s$ ADT*,* DNF $\not\leq_s$ ADT*,* $\texttt{OBDD}_<$ $\not\leq_s$ ADT*,* $\texttt{d-DNNF}_T$ $\not\leq_s$ ADT*, and* EADT $<_s$ ADT*.*

Based on these results, it turns out that $\texttt{OBDD}_<$ does not dominate ADT from the KC point of view (i.e., it does not offer any query/transformation not supported by ADT, and is not strictly more succinct than ADT). This, together with the fact that ADT $\subseteq$ EADT implies that $\texttt{OBDD}_<$ does not dominate EADT. Our succinctness results also reveal that none of the "flat" languages CNF and DNF is at least as succinct as any of ADT or EADT. Although we ignore how d-DNNF and EADT compare w.r.t. succinctness, we know that the subclass $\texttt{d-DNNF}_T$ does not dominate any of ADT or EADT.

4 A CNF-to-EADT Compiler

A natural approach for compiling an arbitrary propositional formula into an EADT formula is to exploit a generalized form of Shannon expansion. Given two formulae Δ and δ, and a variable x, we denote by $\Delta\ |_{x\leftarrow\delta}$ the formula obtained by

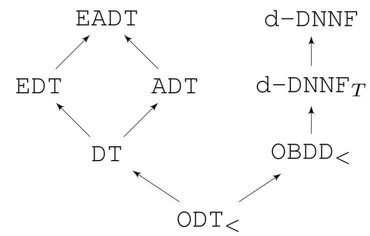

Figure 2: Inclusion graph. $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ indicates that $\mathcal{L}_1 \subseteq \mathcal{L}_2$.

replacing every occurrence of x in Δ by δ. With this notation in hand, the *generalized Shannon expansion* states that for every propositional formulae Δ and δ and every variable x, we have:

$$\Delta \equiv ((x \Leftrightarrow \neg\delta) \wedge \Delta\ |_{x\leftarrow\neg\delta}) \vee ((x \Leftrightarrow \delta) \wedge \Delta\ |_{x\leftarrow\delta})$$

Observe that the standard expansion introduced by Shannon [1949] is recovered by considering $\delta = \top$. The validity of the generalized expansion comes from the fact that Δ is equivalent to $((x \Leftrightarrow \neg\delta) \wedge \Delta) \vee ((x \Leftrightarrow \delta) \wedge \Delta)$, and the fact that, for every propositional formula δ (or its negation), the expression $(x \Leftrightarrow \delta) \wedge \Delta$ is equivalent to $(x \Leftrightarrow \delta) \wedge \Delta\ |_{x\leftarrow\delta}$.

In our setting, Δ is an ECNF formula and δ is an affine clause. ECNF, the language of extended CNF, is the set of all finite conjunctions of extended clauses, where an extended clause is a finite disjunction of affine clauses. Thus, for instance, $x_1 \vee (x_2 \oplus x_3 \oplus \top) \vee (x_1 \oplus x_3)$ is an extended clause. Clearly, CNF is linearly translatable into ECNF Now, since $x \Leftrightarrow \neg\delta$ is equivalent to $x \oplus \delta$, and $x \Leftrightarrow \delta$ is equivalent to $x \oplus \delta \oplus \top$, the generalized Shannon expansion can be restated as the following branching rule:

$$\Delta \equiv ((x \oplus \delta) \wedge \Delta\ |_{x\leftarrow\delta\oplus\top}) \vee ((x \oplus \delta \oplus \top) \wedge \Delta\ |_{x\leftarrow\delta})$$

The Compilation Algorithm. Based on previous considerations, Algorithm 1 provides the pseudo-code for the compiler eadt, which takes as input an ECNF formula Δ, and returns as output an EADT formula equivalent to Δ. The first two lines deal with the specific cases where Δ is valid or unsatisfiable. In both cases, the corresponding leaf is returned.

We note in passing that the unsatisfiability problem (resp. the validity problem) for ECNF has the same complexity as for its subset CNF, i.e., it is coNP-complete (resp. it is in P). This is obvious for the unsatisfiability problem. For the validity problem, an ECNF formula is valid iff every extended clause in it is valid, and an extended clause $\delta_1 \vee \ldots \vee \delta_k$ (where each δ_i, $i \in 1, \ldots, k$ is an affine clause) is valid iff the affine formula $(\delta_1 \oplus \top) \wedge \ldots \wedge (\delta_k \oplus \top)$ is contradictory, which can be tested in polynomial time.

In the remaining case, Δ is split into a decomposable conjunction of components $\Delta_1, \cdots, \Delta_k$ (Line 3). These components are recursively compiled into EADT formulae and conjoined as a $\wedge$ node using the aNode function. Note that decomposition takes precedence over branching: only when Δ consists of a single component, the compiler chooses an affine clause $x \oplus \delta$ for which all variables occur in Δ (Line 5), and then branches on this clause using the generalized Shannon

Algorithm 1: $\mathsf{eadt}(\Delta)$

input : an ECNF formula Δ
output: an EADT formula equivalent to Δ

1 if $\Delta \equiv \top$ then return $\mathsf{leaf}(\top)$
2 if $\Delta \equiv \bot$ then return $\mathsf{leaf}(\bot)$
3 let $\Delta_1, \cdots, \Delta_k$ be the connected components of Δ
4 if $k > 1$ then return $\mathsf{aNode}(\mathsf{eadt}(\Delta_1), \cdots, \mathsf{eadt}(\Delta_k))$
5 choose a simplified affine clause $x \oplus \delta$ such that

$$var(x \oplus \delta) \subseteq Var(\Delta)$$

6 return $\mathsf{dNode}(x \oplus \delta, \mathsf{eadt}(\Delta \mid_{x \leftarrow \delta \oplus \top}), \mathsf{eadt}(\Delta \mid_{x \leftarrow \delta}))$

expansion (Line 6). Here, the dNode function returns a decision node labeled with the first argument, having the second argument as left child, and having the third argument as right child. When Line 3 is omitted, the CNF-to-EADT compiler boils down to a CNF-to-ADT compiler.

Algorithm 1 is guaranteed to terminate. Indeed, by definition of a simplified affine clause, δ in $x \oplus \delta$ is an affine clause which does not contain x. Since none of $\Delta \mid_{x \leftarrow \delta \oplus \top}$ and $\Delta \mid_{x \leftarrow \delta}$ contains x, the steps 5 and 6 can be applied only a finite number of times. Furthermore, the EADT formula returned by the algorithm is guaranteed to satisfy the affine decomposition rule. This property can be proved by induction on the structure of the resulting tree: the only non-trivial case is when the tree consists of a $\wedge$ node with parents N and children $N_1, \cdots, N_k$ each N_i formed by calling eadt on the connected component Δ_i of the formula Δ. Since each parent clause in N is of the form $x \oplus \delta$, where x is excluded from δ, and since the components do not share any variable, it follows that $x \oplus \delta$ overlaps with at most one component in $\Delta_1, \cdots, \Delta_k$. This, together with the fact that the generalized Shannon expansion is valid, establishes the correctness of the algorithm.

Implementation. Algorithm 1 was implemented on top of the state-of-the-art SAT solver MiniSAT [Eén and Sörensson, 2003]. We extended MiniSAT to deal with ECNF formulae. The heuristic used at Line 5 for choosing affine clauses of the form $x \oplus \delta$ is based on the concept of variable activity (VSIDS, Variable State Independent Decaying Sum) [Moskewicz *et al.*, 2001]. Specifically, for each extended clause C of Δ, the score of C is computed as the sum of the scores of each affine clause in it, where the score of an affine clause is the the sum of the VSIDS scores of its variables. Based on this metric, an extended clause C of Δ of maximal score is selected, and the variables of C are sorted by decreasing VSIDS score; the selected variable x is the first variable in the resulting list, and the affine clause δ is formed by the next $k - 1$ variables in the list. Note that selecting all the variables of $x \oplus \delta$ from the same extended clause C of Δ prevents us from generating connections between variables which are not already connected in the constraint graph of Δ. We also took advantage of a simple filtering method, which consists in finding implied affine clauses, used only at the first top nodes of the search tree. In our experiments, we bounded the size of affine clauses to $k = 2$, and used the filtering method up to depth 5.

5 Experiments

Setup. The empirical protocol we followed is very close to the one conducted in [Schrag, 1996] (and other papers). We have considered a number of CNF benchmark instances from different domains provided by the SAT LIBrary (www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html). For each CNF instance Δ, we generated 1000 queries; each query is a 3-literal term γ the 3 variables of which are picked up at random from the set of variables of Δ, following a uniform distribution; the sign of each literal is also selected at random with probability $\frac{1}{2}$. The objective is to count the number of models of the conditioned formula $\Delta \mid \gamma$ for all queries γ. Our experiments have been conducted on a Quad-core Intel XEON X5550 with 32Gb of memory. A time-out of 3 hours has been considered for the off-line compilation phase, and a time-out of 3 hours per query has been established for addressing each of the 1000 queries during the on-line phase. Based on this setup, three approaches have been examined:

- A direct, uncompiled approach: we considered a state-of-the-art model counter, namely Cachet (www.cs.rochester.edu/~kautz/Cachet/index.htm) [Sang *et al.*, 2004]. Here, $\#F_{\mathsf{Cachet}}$ is the number of elements of F_{Cachet}, the set of "feasible" queries, i.e., the queries in the sample for which Cachet has been able to terminate before the time-out (or a segmentation fault). $\overline{Q}_{\mathsf{Cachet}}$ gives the mean time needed to address the feasible queries, i.e.,

$$\overline{Q}_{\mathsf{Cachet}} = \frac{1}{\#F_{\mathtt{Cachet}}} \sum_{\gamma \in F_{\mathsf{Cachet}}} Q_{\mathsf{Cachet}}(\Delta|\gamma)$$

  where $Q_{\mathsf{Cachet}}(\Delta|\gamma)$ is the runtime of Cachet for $\Delta|\gamma$.

- Two compilation-based approaches: d-DNNF and EADT have been targeted. We took advantage of the c2d compiler (reasoning.cs.ucla.edu/c2d/) to generate (smooth) d-DNNF compilations,[2] and our own compiler to compute EADT compiled forms. For each $\mathcal{L}$ among d-DNNF and EADT, Δ has been first turned into a compiled form $\Delta^* \in \mathcal{L}$ during an off-line phase. The compilation time $C_{\mathcal{L}}$ needed to compute Δ^* and the mean query-answering time $\overline{Q}_{\mathcal{L}}$ have been measured. We also computed for each approach two ratios:

$$\alpha_{\mathcal{L}} = \frac{\overline{Q}_{\mathcal{L}}}{\overline{Q}_{\mathtt{Cachet}}} \text{ and } \beta_{\mathcal{L}} = \left\lceil \frac{C_{\mathcal{L}}}{\overline{Q}_{\mathtt{Cachet}} - \overline{Q}_{\mathcal{L}}} \right\rceil$$

  Intuitively, $\alpha_{\mathcal{L}}$ indicates how much on-line time improvement is got from compilation: the lower the better. The quantity $\beta_{\mathcal{L}}$ captures the number of queries needed to amortize compilation time. Clearly, the compilation-based approach targeting $\mathcal{L}$ is useful only if $\alpha_{\mathcal{L}} < 1$. By convention, $\beta_{\mathcal{L}} = +\infty$ when $\alpha_{\mathcal{L}} \geq 1$.

[2] Primarily, we also planned to use the d-DNNF compiler Dsharp [Muise *et al.*, 2012] but unfortunately, we encountered the same problems as mentioned in [Voronov, 2013] to run it, which prevented us from doing it.

Instance			Cachet		c2d				eadt			
name	$\#var$	$\#cla$	$\#F$	$\overline{Q}$	C	$\overline{Q}$	α	β	C	$\overline{Q}$	α	β
ais6	61	581	1000	0.531	1.23	4 E−5	8 E−7	2	0.01	< 1 E−7	< 2 E−7	**1**
ais8	113	1520	866	0.540	3.04	2 E−4	3 E−4	5	0.24	1 E−5	2 E−5	**1**
ais10	181	3151	325	0.578	12.3	1 E−3	2 E−3	21	7.69	1 E−4	2 E−4	**13**
ais12	265	5666	80	0.573	-	-	-	-	410	1 E−3	2 E−3	**717**
bmc-ibm-2	2810	11683	1000	0.569	-	-	-	-	0.37	2 E−5	4 E−5	**1**
bmc-ibm-3	14930	72106	999	13.04	412	0.93	7 E−2	34	180	6 E−3	5 E−4	**13**
bmc-ibm-4	28161	139716	1000	5.412	1128	9.09	1.679	$+\infty$	-	-	-	-
bw_large.a	459	4675	1000	0.537	15.05	2 E−5	4 E−5	28	< 1 E−4	< 1 E−7	< 2 E−7	**1**
bw_large.b	1087	13772	1000	0.612	48.88	5 E−5	8 E−5	79	0.01	< 1 E−7	< 2 E−7	**1**
bw_large.c	3016	50457	996	2.452	283.3	1 E−4	6 E−5	115	0.16	1 E−5	4 E−6	**1**
bw_large.d	6325	131973	896	31.44	-	-	-	-	1.9	2 E−5	6 E−7	**1**
(bw) medium	116	953	1000	0.526	2.39	2 E−5	4 E−5	4	< 1 E−4	< 1 E−7	< 2 E−7	**1**
(bw) huge	459	7054	1000	0.543	15.11	2 E−5	4 E−5	27	< 1 E−4	< 1 E−7	< 2 E−7	**1**
hanoi4	718	4934	505	0.557	559.6	3 E−5	5 E−5	1004	0.13	< 1 E−7	< 2 E−7	**1**
hanoi5	1931	14468	440	0.619	2240	8 E−5	1 E−4	3621	1.1	1 E−5	2 E−5	**1**
logistics.a	828	6718	993	1.266	-	-	-	-	6757	2.12	1.676	$+\infty$
ssa7552-038	1501	3575	1000	0.634	20.99	0.042	0.065	**35**	-	-	-	-

Table 3: Some experimental results

Results. Table 3 presents the obtained results. Each line corresponds to a CNF instance Δ identified by the leftmost column. The first two columns give respectively the number $\#var$ of variables of Δ and the number $\#cla$ of clauses of Δ, and the remaining columns give the measured values. The reported computation times are in seconds.

We can observe that both compilation-based approaches typically prove valuable whenever the off-line compilation phase terminates. On the one hand, for each compilation-based approach $\mathcal{L}$, all the 1000 queries have been "feasible" when the compilation process terminated in due time. For this reason, we did not report in the table the number $\#F_{\mathcal{L}}$ of feasible queries. By contrast, the number of feasible queries for the direct, uncompiled approach is sometimes significantly lower than 1000, and the standard deviation of the on-line query-answering time (not given in the table for space reasons) for such queries is often significantly greater than the corresponding deviations measured from compilation-based approaches. On the other hand, the number of queries β to be considered for balancing the compilation time is finite for all, but two (one for d-DNNF and one for EADT), instances.

Furthermore, the experiments revealed that some instances of significant size are compilable. When the compilation succeeds, β is typically small and, accordingly, on-line time savings of several orders of magnitude can be achieved. Especially, the optimal value 1 for the "break-even" point β has been reached for many instances when the EADT language was targeted. This means that in many cases the off-line time spend to built the EADT compiled form is immediately balanced by the first counting query. Stated otherwise, the eadt compiler proves also competitive as a model counter.

Finally, our experiments show EADT compilation challenging with respect to d-DNNF compilation in many (but not all) cases. When compilation succeeds in both cases, the number of nodes in EADT and d-DNNF formulae are about the same order, but the EADT formulae are slightly faster for processing queries, due to their arborescent structure.

6 Conclusion

The propositional language EADT introduced in the paper appears as quite appealing for the representation purpose, when **CT** is a key query. Especially, EADT offers the same queries as d-DNNF, and more transformations (among those considered in the KC map). The subset ADT of EADT offers all the queries and the same transformations as those satisfied by the influential $\text{OBDD}_<$ language. Furthermore, $\text{OBDD}_<$ is not at least as succinct as ADT, which shows ADT as a possible challenger to $\text{OBDD}_<$. In practice, the EADT compilation-based approach to model counting appears as competitive with the model counter Cachet and the d-DNNF compilation-based approach to model counting.

This work opens a number of perspectives for further research. From the theoretical side, a natural extension of ADT is the set of all single-rooted finite DAGs, where leaves are labeled by a Boolean constant ($\top$ or $\bot$), and internal nodes are affine decision nodes. However, this language is not appealing as a target language for knowledge compilation, because it contains the language BDD of binary decision diagrams (alias branching programs) [Bryant, 1986] as a subset and BDD does not offer *any* query from the KC map [Darwiche and Marquis, 2002], unless $\mathsf{P} = \mathsf{NP}$. Thus, the problem of finding interesting classes of affine decision graphs that are tractable for model counting looks stimulating.

From the practical side, there are many ways to improve our compiler. Notably, it would be interesting to take advantage of preprocessing techniques [Piette *et al.*, 2008; Järvisalo *et al.*, 2012] in order to simplify the input CNF formulae before compiling them. Furthermore, it could prove useful to exploit Gaussian elimination for handling more efficiently (see e.g. [Li, 2003; Chen, 2007; Soos *et al.*, 2009]) instances that contain subproblems corresponding to affine formulae, like those reported in [Crawford and Kearns, 1995; Cannière, 2006]. Finally, considering other heuristics for selecting the branching affine clauses (e.g. criteria based on the mutual information metric) could also prove valuable.

References

[Bacchus *et al.*, 2003] F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #SAT and bayesian inference. In *Proc. of FOCS'03*, pages 340–351, 2003.

[Bordeaux *et al.*, 2012] L. Bordeaux, M. Janota, J. P. Marques Silva, and P. Marquis. On unit-refutation complete formulae with existentially quantified variables. In *Proc. of KR'12*, 2012.

[Bryant, 1986] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–692, 1986.

[Cannière, 2006] Ch. De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In *Proc. of ISC'06*, pages 171–186, 2006.

[Chen, 2007] J. Chen. XORSAT: An efficient algorithm for the dimacs 32-bit parity problem. *CoRR*, abs/cs/0703006, 2007.

[Crawford and Kearns, 1995] J.M. Crawford and M.J. Kearns. The minimal disagreement parity problem as a hard satisfiability problem. Technical report, 1995.

[Darwiche and Marquis, 2002] A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[Darwiche, 2001] A. Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.

[Darwiche, 2009] Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

[Darwiche, 2011] A. Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Proc. of IJCAI'11*, pages 819–826, 2011.

[Eén and Sörensson, 2003] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. of SAT'03*, pages 502–518. 2003.

[Fargier and Marquis, 2008] H. Fargier and P. Marquis. Extending the knowledge compilation map: Krom, Horn, affine and beyond. In *Proc. of AAAI'08*, pages 442–447, 2008.

[Gergov and Meinel, 1994] J. Gergov and C. Meinel. Efficient analysis and manipulation of OBDDs can be extended to FBDDs. *IEEE Transactions on Computers*, 43(10):1197–1209, 1994.

[Järvisalo *et al.*, 2012] M. Järvisalo, M. Heule, and A. Biere. Inprocessing rules. In *Proc. of IJCAR'12*, pages 355–370, 2012.

[Li, 2003] C. Li. Equivalent literal propagation in the dll procedure. *Discrete Applied Mathematics*, 130(2):251–276, 2003.

[Littman *et al.*, 2001] M. L. Littman, S. M. Majercik, and T. Pitassi. Stochastic boolean satisfiability. *Journal of Automated Reasoning*, 27(3):251–296, 2001.

[Marquis, 2011] P. Marquis. Existential closures for knowledge compilation. In *Proc. of IJCAI'11*, pages 996–1001, 2011.

[Mateescu *et al.*, 2008] R. Mateescu, R. Dechter, and R. Marinescu. AND/OR multi-valued decision diagrams (AOMDDs) for graphical models. *Journal of Artificial Intelligence Research*, 33:465–519, 2008.

[Moskewicz *et al.*, 2001] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proc. of DAC'01*, pages 530–535, 2001.

[Muise *et al.*, 2012] Ch.J. Muise, Sh.A. McIlraith, J.Ch. Beck, and E.I. Hsu. Dsharp: Fast d-DNNF compilation with sharpSAT. In *Proc. of AI'12*, pages 356–361, 2012.

[Piette *et al.*, 2008] C. Piette, Y. Hamadi, and L. Saïs. Vivifying propositional clausal formulae. In *Proc. of ECAI'08*, pages 525–529, 2008.

[Pipatsrisawat and Darwiche, 2008] K. Pipatsrisawat and A. Darwiche. New compilation languages based on structured decomposability. In *Proc. of AAAI'08*, pages 517–522, 2008.

[Pipatsrisawat and Darwiche, 2010] K. Pipatsrisawat and A. Darwiche. A lower bound on the size of decomposable negation normal form. In *Proc. of AAAI'10*, 2010.

[Roth, 1996] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1–2):273–302, 1996.

[Sang *et al.*, 2004] T. Sang, F. Bacchus, P. Beame, H.A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Proc. of SAT'04*, 2004.

[Sang *et al.*, 2005] T. Sang, P. Beame, and H. A. Kautz. Performing Bayesian inference by weighted model counting. In *Proc. of AAAI'05*, pages 475–482, 2005.

[Schaefer, 1978] Th. J. Schaefer. The complexity of satisfiability problems. In *Proc. of STOC'78*, pages 216–226, 1978.

[Schrag, 1996] R. Schrag. Compilation for critically constrained knowledge bases. In *Proc. of AAAI'96*, pages 510–515, 1996.

[Shannon, 1949] C.E. Shannon. The synthesis of two–terminal switching circuits. *Bell System Technical Journal*, 28(1):59–98, 1949.

[Soos *et al.*, 2009] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *Proc. of SAT'09*, pages 244–257, 2009.

[Subbarayan *et al.*, 2007] S. Subbarayan, L. Bordeaux, and Y. Hamadi. Knowledge compilation properties of tree-of-BDDs. In *Proc. of AAAI'07*, pages 502–507, 2007.

[Valiant, 1979] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[Voronov, 2013] A. Voronov. *On Formal Methods for Large-Scale Product Configuration*. Ph.D. thesis, Chalmers University, 2013.

[Wachter and Haenni, 2006] M. Wachter and R. Haenni. Propositional DAGs: A new graph-based language for representing Boolean functions. In *Proc. of KR'06*, pages 277–285, 2006.

Appendix

Proof:[Proposition 1] We start with the queries. Concerning **CT**, the proof is given in Section 3. For the remaining queries, we take advantage of the obvious polynomial translation existing between any language and a superset of it. Thus whenever EADT (resp. ADT) satisfies a given query, this query is also satisfied by EDT, ADT, DT, and ODT$_<$ (resp. DT, and ODT$_<$). Conversely, whenever a query is NP-hard for EDT, it is also NP-hard for EADT.

- **CO**, **VA**: comes directly from the fact that EADT satisfies **CT**.
- **CE**, **ME**: comes directly from the fact that EADT satisfies **CO** and **CD** (Lemmata A.3 and A.4 from [Darwiche and Marquis, 2002] can be easily extended to ADN formulae).
- **IM**: comes directly from the fact that EADT satisfies **VA** and **CD** (Lemma A.7 from [Darwiche and Marquis, 2002] can be easily extended to ADN formulae).
- **SE**: for ADT, the result comes directly from the fact that ADT satisfies $\wedge$**BC**, $\neg$**C** and **CO**. Indeed, for any ADN formulae α and β, we have $\alpha \models \beta$ iff $\alpha \wedge \neg\beta$ is inconsistent. For the class EDT we take advantage of the same reduction as the one used to prove that EDT does not satisfy $\wedge$**BC**. Indeed, the problem of determining whether the conjunction of two EDT formulae α_1 and α_2 is consistent is NP-complete. Since $\alpha_1 \wedge \alpha_2$ is consistent iff $\alpha_1 \not\models \neg\alpha_2$ and since EDT satisfies $\neg$**C**, the hardness result follows.
- **EQ**: comes directly from the fact that ADT satisfies **SE**, since for any ADN formulae α and β, we have $\alpha \equiv \beta$ iff $\alpha \models \beta$ and $\beta \models \alpha$.

We continue with the transformations:

- **CD**: let γ be a consistent term. If α is an EADT (resp. ADT, EDT, DT, ODT$_<$) formula, the tree obtained by replacing every affine clause δ labeling a node of α by $\delta \mid \gamma$ is an EADT (resp. ADT, EDT, DT, ODT$_<$) formula equivalent to $\alpha \mid \gamma$. Indeed, the conditioning transformation "distributes" over the connectives; especially:
  - $(\Delta_1 \wedge \ldots \wedge \Delta_k) \mid \gamma = (\Delta_1 \mid \gamma) \wedge \ldots \wedge (\Delta_k \mid \gamma)$.
  - $(\Delta_1 \vee \ldots \vee \Delta_k) \mid \gamma = (\Delta_1 \mid \gamma) \vee \ldots \vee (\Delta_k \mid \gamma)$.

  Thus, when we have
  $$\Delta \equiv ((x \oplus \delta) \wedge \Delta[x \leftarrow \delta \oplus \top]) \\ \vee((x \oplus \delta \oplus \top) \wedge \Delta[x \leftarrow \delta]),$$
  then $\Delta \mid \gamma$ is equivalent to
  $$((x \oplus \delta) \mid \gamma \wedge \Delta[x \leftarrow \delta \oplus \top] \mid \gamma) \\ \vee((x \oplus \delta \oplus \top) \mid \gamma \wedge \Delta[x \leftarrow \alpha] \mid \gamma).$$
  Observe that $(x \oplus \delta) \mid \gamma$ is an affine clause when δ is an affine clause and that $(x \oplus \delta \oplus \top) \mid \gamma$ is equivalent to $((x \oplus \delta) \mid \gamma) \oplus \top$. As to the case of EDT, observe also that for any literal l, $l \mid \gamma$ is equivalent to l, or $\top$ or $\bot$, which are allowed labels for the decision nodes of an EDT formula). Finally, observe that when α is an ODT$_<$ formula, the ordering constraint induced by $<$ is preserved by such a conditioning process. The fact that the replacement process can obviously be achieved in time linear in the size of α plus the size of γ completes the proof.
- **FO**: by reduction from the validity problem for DNF formulae. Let $\alpha = \bigvee_{i=1}^{m} \gamma_i$ where each γ_i ($i \in 1, \ldots, m$) is a term. We associate in polynomial time with α the DNF formula β given by $\bigvee_{i=1}^{m}(\gamma_i \wedge \bigwedge_{j=1}^{i-1} \neg new_j \wedge new_i)$ where $\{new_1, \ldots, new_m\}$ are fresh variables from PS, not occurring in α. By construction, β is a deterministic DNF formula (i.e., its terms are pairwise inconsistent) and we have $\alpha \equiv \exists\{new_1, \ldots, new_m\}.\beta$. Now, we can easily associate with β in polynomial time an equivalent ODT$_<$ formula $\beta^* =$
  $$\langle new_1, \langle new_2, \ldots, \\ \langle new_{m-1}, \langle new_m, \bot, \gamma_m^*\rangle, \gamma_{m-1}^*\rangle \ldots, \gamma_1^*\rangle,$$
  where each γ_i^* ($i \in 1, \ldots, m$) is an ODT$_<$ formula equivalent to the term γ_i. Basically, whatever the $<$ ordering over $Var(\alpha)$ which is considered, the size of γ_i^* is linear in the size of γ_i. And β^* respects the extension of $<$ over $Var(\alpha) \cup \{new_1, \ldots, new_m\}$ given by $\forall i \in 1, \ldots, m-1, new_i < new_{i+1}$ and $\forall i \in 1, \ldots, m$, $\forall x \in Var(\alpha), new_i < x$.
  Therefore, we have $\alpha \equiv \exists\{new_1, \ldots, new_m\}.\beta^*$. Suppose that ODT$_<$ (resp. DT, EDT, ADT, EADT) would satisfy **FO**. Then we could compute in time polynomial in the size of β^* an ODT$_<$ formula (hence a DT, EDT, ADT, EADT formula) equivalent to $\exists\{new_1, \ldots, new_m\}.\beta^*$, hence equivalent to α. Since each of ODT$_<$, DT, EDT, ADT, EADT satisfies **VA**, we could determine in time polynomial in the size of α whether α is valid. As a consequence, we would have coNP = P, hence P = NP.
- **SFO**: as to ADT (resp. DT, ODT$_<$), the result comes directly from the fact that ADT (resp. DT, ODT$_<$) satisfies **CD** and $\vee$**BC**. Indeed, for every propositional formula α and variable x, we have $\exists x.\alpha \equiv (\alpha \mid \neg x) \vee (\alpha \mid x)$.
  As to EDT (resp. EADT), let us consider two EDT (resp. EADT) formulae α_1 and α_2 and let x be a fresh variable, i.e., $x \notin Var(\alpha_1) \cup Var(\alpha_2)$. The EDT (resp. EADT) formula $\beta = \langle x, \alpha_1, \alpha_2\rangle$ can be computed in time linear in the size of α_1 plus the size of α_2, and by construction it is such that $\alpha_1 \vee \alpha_2 \equiv \exists x.\beta$. If EDT (resp. EADT) would satisfy **SFO**, then an EDT (resp. EADT) formula equivalent to $\alpha_1 \vee \alpha_2$ could be generated in time polynomial in the size of α_1 plus the size of α_2, just by forgetting x in β. As a consequence, EDT (resp. EADT) would satisfy $\vee$**BC**. This is not the case unless P = NP.
- $\wedge$**C**: by reduction from the satisfiability problem for CNF formulae. Let $\alpha = \bigwedge_{i=1}^{m} \delta_i$ where each δ_i ($i \in 1, \ldots, m$) is a clause. With each δ_i ($i \in 1, \ldots, m$) we can associate in linear time an ODT$_<$ formula δ_i^*

equivalent to δ_i. Suppose that $\texttt{ODT}_<$ (resp. DT, EDT, ADT, EADT) would satisfy $\wedge\mathbf{C}$. Then we could compute in time polynomial in the size of α an $\texttt{ODT}_<$ formula (hence a DT, EDT, ADT, EADT formula) α^* equivalent to $\bigwedge_{i=1}^{m} \delta_i^*$, hence equivalent to α. Since each of $\texttt{ODT}_<$, DT, EDT, ADT, EADT satisfies **CO**, we could determine in time polynomial in the size of α whether α is consistent. As a consequence, we would have P = NP.

- $\wedge\mathbf{BC}$: as to EDT (resp. EADT), the proof is by reduction from the satisfiability problem for CNF formulae. Consider a CNF formula $\alpha = \bigwedge_{i=1}^{m} \delta_i$ over $Var(\alpha) = \{x_1, \ldots, x_n\}$, where each δ_i ($i \in 1, \ldots, m$) is a clause. Associate with α in polynomial time the CNF formulae α_1 and α_2 such that $\alpha_1 = \bigwedge_{i=1}^{m} \delta_i^i$ where each δ_i^i ($i \in 1, \ldots, m$) is the clause δ_i in which every variable x_j ($j \in 1, \ldots, n$) is replaced by a fresh variable x_j^i, and $\alpha_2 = \bigwedge_{j=1}^{n}(\bigwedge_{i=1}^{m}((\neg x_j \vee x_j^i) \wedge (x_j \vee \neg x_j^i)))$. By construction, α is satisfiable iff $\alpha_1 \wedge \alpha_2$ is satisfiable. The point is that α_1 (resp. α_2) can be associated in polynomial time with an equivalent EDT formula (hence an EADT formula) α_1^* (resp. α_2^*). Indeed, by construction, the outermost $\bigwedge$ connectives in α_1 and in α_2 are decomposable. Thus we can consider the EDT formula $\alpha_1^* = \bigwedge_{i=1}^{m} \delta_i^{i*}$ where each δ_i^{i*} ($i \in 1, \ldots, m$) is a DT representation of the clause δ_i^i (and we know that such DT representations can be generated in time linear in the size of the corresponding clause.) Similarly, α_2 is equivalent to the EDT formula (hence an EADT formula) $\alpha_2^* = \bigwedge_{j=1}^{n} \beta_j$ where each β_j ($j \in 1, \ldots, n$) is a DT representation of $\bigwedge_{i=1}^{m}((\neg x_j \vee x_j^i) \wedge (x_j \vee \neg x_j^i))$, which is equivalent to the DNF formula $(x_j \wedge \bigwedge_{i=1}^{m} x_j^i) \vee (\neg x_j \wedge \bigwedge_{i=1}^{m} \neg x_j^i)$. Thus β_j ($j \in 1, \ldots, n$) can be taken as equal to

$$\langle x_j, \langle x_j^1, \langle x_j^2, \ldots, \langle x_j^m, \top, \bot\rangle, \bot\rangle, \bot\rangle \ldots\rangle,$$

$$\langle x_j^1, \bot, \langle x_j^2, \bot, \langle \ldots, \langle x_j^m, \bot, \top\rangle, \ldots\rangle.$$

Now, if any of EDT or EADT would satisfy $\wedge\mathbf{BC}$, then for testing the satisfiability of α, it would be enough to compute both α_1^* and α_2^* in time polynomial in the size of α, then compute an EDT formula (or an EADT formula) α^* equivalent to $\alpha_1^* \wedge \alpha_2^*$, and finally test the satisfiability of α^* in polynomial time since each of EDT and EADT satisfies **CO**. This would make P = NP.

As to ADT (resp. DT), given two ADT (resp. DT) formulae α_1 and α_2, an ADT (resp. DT) formula α equivalent to $\alpha_1 \wedge \alpha_2$ can be obtained by replacing in α_1 every leaf node labeled by $\top$ by a copy of α_2. Clearly enough, this can be achieved in quadratic time in the size of α_1 plus the size of α_2. Since $T(\alpha)$ is equivalent to $T(\alpha_1) \wedge T(\alpha_2)$, we get that α is equivalent to $\alpha_1 \wedge \alpha_2$. In the DT case, it is not guaranteed that the resulting tree α is read-once.

As to $\texttt{ODT}_<$, compute first the ADN formula α equivalent to $\alpha_1 \wedge \alpha_2$ as explained for the ADT case above. In the general case, such a formula α is not an $\texttt{ODT}_<$ formula because some paths from the root of α to a leaf may contain two decision nodes labeled by the same decision variable. However, it is easy to turn α into an equivalent $\texttt{ODT}_<$ formula in linear time, via a reduction process. First of all, once can easily turn α in linear time into an equivalent tree such that every decision node N is labeled by a unary affine clause which has the form of a variable x. Indeed, if N is labeled by $\top$ (resp. $\bot$) then by-pass it by connecting its father to N_+ (resp. N_-) (of course, if N is the root of the tree, just replace it by N_+ (resp. N_-)). If N is labeled by a unary affine clause of the form $\neg x$, then switch its two children N_+ and N_-. Once this is done, for every path p from the root of the resulting tree to a leaf containing two decision nodes labeled by the same decision variable x, if the first node N in p among those labeled by x is connected to N^+ (resp. N^-), then by-pass every other node N' labeled by x in p by connecting its father to N'^+ (resp. N'^-); repeat this reduction rule until no such path p can be found. By construction, the resulting formula is an $\texttt{ODT}_<$ formula equivalent to α. Note that if this reduction process is applied to a DT formula, the resulting DT formula is read-once.

- $\vee\mathbf{C}$: comes directly from the fact that none of $\texttt{ODT}_<$, DT, EDT, ADT, and EADT satisfies $\wedge\mathbf{C}$ unless P = NP and each of them satisfies $\neg\mathbf{C}$ (simply use De Morgan laws).

- $\vee\mathbf{BC}$: as to $\texttt{ODT}_<$ (resp. DT, ADT), the result comes directly from the fact that $\texttt{ODT}_<$ (resp. DT, ADT) satisfies $\wedge\mathbf{BC}$ and $\neg\mathbf{C}$. As to EDT and EADT, the result comes directly from the fact that each of EDT and EADT satisfies $\neg\mathbf{C}$ but none of them satisfies $\wedge\mathbf{BC}$ unless P = NP. In both cases, take advantage of De Morgan laws.

- $\neg\mathbf{C}$: we explain how any EADT formula α can be associated in linear time with an EADT formula $n(\alpha)$ equivalent to $\neg\alpha$. n proceeds recursively as follows, starting with $N = R_\alpha$:
  - if $N = \langle\delta, N_-, N_+\rangle$, then $n(N) = \langle\delta, n(N_-), n(N_+)\rangle$;
  - if $N = \bigwedge_{i=1}^{k} N_i$, then $n(N) = \bigvee_{i=1}^{k} n(N_i)$;
  - if $N = \bigvee_{i=1}^{k} N_i$, then $n(N) = \bigwedge_{i=1}^{k} n(N_i)$;
  - if $N = \top$, then $n(N) = \bot$;
  - if $N = \bot$, then $n(N) = \top$.

Observe that in the cases $N = \bigwedge_{i=1}^{k} N_i$ and $N = \bigvee_{i=1}^{k} N_i$, the n transformation preserves the decomposability of the nodes. That $n(N)$ is equivalent to $\neg I(N)$ is obvious for every case, except for the case when $N = \langle\delta, N_-, N_+\rangle$. In this latter case, we know that $I(N) = ((\delta \oplus \top) \wedge I(N_-)) \vee (\delta \wedge I(N_+))$. Therefore, $\neg I(N) \equiv (\delta \vee \neg I(N_-)) \wedge ((\delta \oplus \top) \vee \neg I(N_+))$. After distribution and simplification, we obtain $((\delta \oplus \top) \wedge \neg I(N_-)) \vee (\delta \wedge \neg I(N_+))$. Thus, by structural induction, $\neg I(N)$ is equivalent to $((\delta \oplus \top) \wedge n(N_-)) \vee (\delta \wedge n(N_+))$. Equivalently, $\neg I(N)$ is equivalent to $\langle\delta, n(N_-), n(N_+)\rangle$.

Finally, when α is an ADT formula (resp. an EDT, DT, $\texttt{ODT}_<$ formula), $n(\alpha)$ also is an ADT formula (resp. an

EDT, DT, $\text{ODT}_<$ formula). In the ADT, DT, and $\text{ODT}_<$ cases, $n(\alpha)$ can be simply obtained from α by replacing in it every $\top$ leaf by a $\bot$ leaf and every $\bot$ leaf by a $\top$ leaf.

■

Proof:[Proposition 2]

- $\text{OBDD}_< \not\leq_s$ ADT: it is known that the circular bit shift function cbs_m does not have a polynomial-sized SDNNF representation [Pipatsrisawat and Darwiche, 2010], hence a polynomial-sized $\text{OBDD}_<$ representation since SDNNF $<_s$ $\text{OBDD}_<$. For any positive integer m, consider the following Boolean function over $2^{m+1}+m$ variables $cbs_m(x_0, \ldots, x_{2^m-1}, y_0, \ldots, y_{2^m-1}, i_0, \ldots, i_{m-1})$ which is the semantics of the formula $\alpha_m =$

  $$\bigvee_{b_0,\ldots,b_{m-1}\in\{0,1\}} (\bigwedge_{j=0}^{m-1} i_j^{b_j} \wedge \bigwedge_{j=0}^{2^m-1} x_j \Leftrightarrow y_{(j+num(b_0,\ldots,b_{m-1}))mod2^m}),$$

  whose size is linear in the number of variables of cbs_m. In this formula, $i_j^{b_j}$ denotes the literal i_j when $b_j = 0$ and the literal $\neg i_j$ when $b_j = 1$; num is the mapping from $\{0,1\}^m$ to the set of natural numbers which gives the integer represented by the binary string $b_0 \ldots b_{m-1}$. Thus, the variables $i_0, \ldots, i_{m-1}$ make precise how the bits of the binary string $y_0 \ldots y_{2^m-1}$ must be (circularly) shifted, and $cbs_m(x_0, \ldots, x_{2^m-1}, y_0, \ldots, y_{2^m-1}, i_0, \ldots, i_{m-1}) = 1$ exactly when the variables $x_0, \ldots, x_{2^m-1}$ and the shifted variables $y_0, \ldots, y_{2^m-1}$ are pairwise equal.

  For each $b_0, \ldots, b_{m-1} \in \{0,1\}$, the formula $\beta_{b_0,\ldots,b_{m-1}} =$

  $$\bigwedge_{j=0}^{2^m-1} x_j \Leftrightarrow y_{(j+num(b_0,\ldots,b_{m-1}))mod2^m}$$

  is equivalent to the AFF formula $\gamma_{b_0,\ldots,b_{m-1}} =$

  $$\bigwedge_{j=0}^{2^m-1} x_j \oplus y_{(j+num(b_0,\ldots,b_{m-1}))mod2^m} \oplus \top.$$

  Clearly enough, $\gamma_{b_0,\ldots,b_{m-1}}$ can be computed in time linear in the size of $\beta_{b_0,\ldots,b_{m-1}}$, hence linear in the number of variables of cbs_m. Since AFF $\geq_p$ ADT, an ADT formula $\delta_{b_0,\ldots,b_{m-1}}$ equivalent to $\beta_{b_0,\ldots,b_{m-1}}$ can also be generated in time polynomial in the size of $\beta_{b_0,\ldots,b_{m-1}}$.

  As a consequence, an ADT formula equivalent to α_m can be generated from α_m in polynomial time by generating first a DT formula with decision nodes labelled by variables from $\{i_0, \ldots, i_{m-1}\}$, and whose 2^m paths correspond to all possible terms of the form $\bigwedge_{j=0}^{m-1} i_j^{b_j}$ with $b_0, \ldots, b_{m-1} \in \{0,1\}$; then each leaf node of this tree associated with $b_0, \ldots, b_{m-1}$ is replaced by the root of $\delta_{b_0,\ldots,b_{m-1}}$. By construction, the size of the resulting ADT formula is linear in the number of variables of cbs_m.

- CNF $\not\leq_s$ ADT and DNF $\not\leq_s$ ADT: comes from the fact that AFF $\geq_p$ ADT(hence ADT $\leq_s$ AFF), while CNF $\not\leq_s$ AFF, and DNF $\not\leq_s$ AFF (because of the "parity function" $\oplus_{i=1}^n x_i$, which is a XOR-clause with no polynomial-sized DNF or CNF representation).

- EADT $<_s$ ADT: from the inclusion ADT $\subseteq$ EADT, we obviously get that EADT $\leq_s$ ADT. Now, consider the CNF formula $\Delta = \bigwedge_{i=1}^n (\neg x_i \vee \neg y_i)$. [Fargier and Marquis, 2008] shows that Δ cannot be represented by an AFF[$\vee$] formula of size polynomial in n. Furthermore, ADT is polynomially translatable into AFF[$\vee$]. Indeed, let α be an ADT formula. Let β be the disjunction, for all paths of α from its root to a $\top$ node, of the conjunction of all affine clauses encountered in the path. β can be generated in time linear in the size of α, and by construction it is a disjunction of affine formulae that is equivalent to α. As a consequence, Δ cannot be represented by an ADT formula of size polynomial in n. Contrastingly, Δ has a polynomial-sized EADT representation (and even a polynomial-sized EDT representation) since the sets of variables of the clauses $\neg x_i \vee \neg y_i$ ($i \in 1, \ldots, n$) of Δ are pairwise disjoint and CL is polynomially translatable into DT.

■