Ordered Functional Decision Diagrams

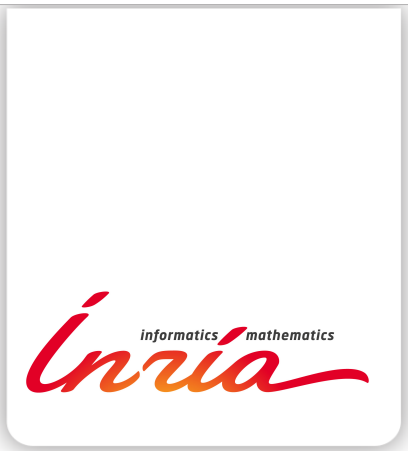
Joan Thibault, Khalil Ghorbal

Ordered Functional Decision Diagrams: A Functional Semantic For Binary Decision Diagrams

Joan Thibault, Khalil Ghorbal

RESEARCH REPORT
N° 9333
Mars 2020
Project-Teams HYCOMES

ISSN 0249-6399 ISRN INRIA/RR--9333--FR+ENG



Ordered Functional Decision Diagrams: A Functional Semantic For Binary Decision Diagrams

Joan Thibault, Khalil Ghorbal

Project-Teams HYCOMES

Research Report n° 9333 — Mars 2020 — 26 pages

Abstract: Several BDD variants were designed to exploit special features of Boolean functions to achieve better compression rates. Deciding *a priori* which variant to use is as hard as constructing the diagrams themselves and the conversion between variants comes in general with a prohibitive cost. This observation leads naturally to a growing interest into when and how one can combine existing variants to benefit from their respective sweet spots. In this paper, we introduce a novel framework, termed λDD, that revisits BDD from a purely functional point of view. The framework allows to classify the already existing variants, including the most recent ones like Chain-DD and ESRBDD, as implementations of a special class of ordered models. We enumerate, in a principled way, all the models of this class and isolate its most expressive model. This new model, termed λDD-O-NUCX, is suitable for both dense and sparse Boolean functions, and, unlike Chain-DD and ESRBDD, is *invariant* by negation. The canonicity of λDD-O-NUCX is formally verified using the Coq proof assistant. We furthermore provide experimental evidence corroborating our theoretical findings: more expressive λDD models achieve, indeed, better memory compression rates.

Key-words: Binary Decision Diagrams, Functional Abstraction

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Diagramme de Décision Fonctionnel Ordonné: Une Sémantique Fonctionnelle pour les Diagrammes de Décision Binaires

Résumé : Les Diagrammes de Décision Binaires (BDD) sont une structure qui a été décliné en de nombreux variants, chacun conçu pour exploiter au mieux une propriété particulière des fonctions Booléennes et ainsi réduire leur taille mémoire. Cependant, décider quel variant est le plus adapté à priori est aussi difficile que construire chaque représentation. De plus la conversion entre les variants coûte également trop chère pour être utilisable en pratique. Ces observations ont conduit naturellement à un intérêt grandissant pour combiner des variants pour tirer parti de leurs avantages respectifs. Cet article introduit un cadre théorique à cette problématique en introduisant une méta-structure nommée λDD qui adopte un point de vue purement fonctionnel sur les BDD. Les diagrammes λDD permettent d'abstraire les variants classiques (par exemple ROBDD et ZDD) ainsi que des variants récents (en particulier ChainDD et ESRBDD) comme des modèles d'une classe de modèles ordonnés que nous appelons λDD-O. Dans ces modèles ordonnés, les variables sont évaluées en suivant un ordre global strict. Nous énumérons les éléments de cette classe et isolons son modèle le plus expressif que nous nommons λDD-O-NUCX. Ce modèle permet non seulement de capturer les variables canalisantes (ce qui le rend plus expressif que ESRBDD par exemple) mais exploite aussi une propriété supplémentaire: les xor-variables. Le modèle est également invariant par négation, ce qui permet de calculer la négation d'une fonction Booléenne en temps constant. La canonicité de λDD-O-NUCX est formellement vérifiée en utilisant l'assistant de preuve Coq. De plus, nous fournissons des résultats expérimentaux confirmant l'intérêt de λDD-O-NUCX.

Mots-clés : Diagramme de Décision Binaire, Abstraction Fonctionelle

Introduction

A Binary Decision Diagram (BDD) is a versatile graph-based data structure, well suited to effectively represent and manipulate Boolean functions. The introduction of Reduced Ordered BDD (ROBDD) by Bryant [1] in 1986 widely contributed to their adoption. Indeed, enforcing a total ordering over the variables limits combinatorial explosion and makes the structure *canonical*, that is, in one-to-one correspondence with Boolean functions.

Even though a ROBDD has an exponential worst-case size, many practical applications yield more concise representations thanks to the detection and elimination of *useless* nodes, i.e., those nodes having their outgoing edges pointing towards the same subgraph. Many other BDD variants [2, 3, 4] have been subsequently designed to capture specific application-dependent properties in order to further reduce the size of the diagram or to efficiently perform specific operations. For instance, Zero-suppressed Decision Diagrams [5, 6], or ZDD, form a notable variant that is well suited to encode sparse functions, i.e., functions that evaluate to zero except for a limited number of valuations of their inputs.

Most recently, two new variants, namely Chain-BDD [7] and ESRBDD [8], propose to combine ROBDD and ZDD in order to get a data structure suitable for both dense and sparse functions. In this work, we are also interested in combining existing variants in order to benefit from their respective sweet spots.

Our approach is however, drastically different: we combine reduction rules by composing their *functional abstraction* (or interpretation). To do so, we introduce a new functional framework, together with its related data structure, that we term λDD. Special variables, like useless variables, are captured by elementary operators (or functors) acting on Boolean functions. We exemplify our approach by considering the so-called *canalizing* variables [9], which form an important class of special variables dual in a sense to useless variables: their valuation fixes the output of the function regardless of the valuation of the other variables. In our framework, designing a data structure that captures several special variables amounts to combining, at the functional level, various elementary operators, while paying attention to their possible interactions.

The functional framework allows not only to compare the expressive power of the modeled variants, but also, and more importantly, to design in a principled way new models with higher compression rates. We present in particular a new canonical data structure, termed λDD-O-NUCX, that combines canalizing and useless variables while supporting negation, unlike ZDD, Chain-BDD, and ESRBDD. Moreover, the obtained graphs are *invariant* by negation, i.e., the diagram of the negation of a function differs from the diagram of the function itself by only appending the symbol that encodes negation. As a consequence, the negation operation on the structure is performed in constant time.

We implement and asses the different models on standard sets of benchmarks. The results presented in the Experiments section consolidate several theoretical intuitions regarding the potential gains in compression rates achieved by expressive models.

Contributions To summarize, our contributions are:

1. A general functional framework for Boolean functions relying on the Shannon combinator (Section 2) for a class of *ordered* models, denoted λDD−O, classifying many already existing BDD variants (Section 4).
2. A new model, called λDD−O−NUCX (Section 5), supporting all the primitives defining the class λDD−O, including negation (Section 3). As such, the model is a strict generalization of the recent variants Chain−DD and ESRBDD. Its canonicity is formally proved in the Coq proof assistant.
3. An implementation and experimental evidence showing the superiority of λDD−O−NUCX in terms of compression rates for both dense and sparse functions (Section 6).

1 Background on Boolean Functions

A Boolean function of arity $n \in \mathbb{N}$ is a form (or functional) from $\mathbb{B}^n$ to $\mathbb{B}$. It operates on an ordered tuple of Booleans of dimension n, $(x_0, \ldots, x_{n-1})$, by assigning a Boolean value to each of the 2^n valuations of its tuple.

The set of Boolean functions of arity n, denoted by $\mathbb{B}^{n\to 1}$, is thus finite and contains 2^{2^n} elements. In particular, $\mathbb{B}^{0\to 1}$ has two elements and is isomorphic to $\mathbb{B}$ itself (only the types differ: functions on the one hand, and co-domain elements, or Booleans, on the other hand). To avoid confusion, we use a different font for functions: *0* will denote the constant function of arity zero returning 0, and *1* will denote the constant function of arity zero returning 1.

We rely on a binary non-commutative operator, sometimes referred to as the Shannon operator in the literature, defined as follows.

Definition 1 (Shannon operator)**.**

$$\begin{aligned} \star : \mathbb{B}^{n\to 1} \times \mathbb{B}^{n\to 1} &\to \mathbb{B}^{n+1\to 1} \\ (f, g) &\mapsto f \star g \end{aligned}$$

where

$$\begin{aligned} f \star g : \mathbb{B}^{n+1} &\to \mathbb{B} \\ (x_0, x_1, \ldots, x_n) &\mapsto \begin{cases} f(x_1, \ldots, x_n) & \text{if } x_0 = 0 \\ g(x_1, \ldots, x_n) & \text{if } x_0 = 1 \end{cases} \end{aligned}$$

Another way to define $f \star g$, using logical connectives, would be

$$(f \star g) : (x_0, x_1, \ldots, x_n) \mapsto (\neg x_0 \wedge f(x_1, \ldots, x_n)) \vee (x_0 \wedge g(x_1, \ldots, x_n)) \ .$$

The Shannon operator is (i) *universal*: any Boolean function can be fully decomposed, by induction over its arity, all the way down to constant functions; and (ii) *elementary*: it operates on two functions of the same arity and increases the arity by exactly one. In our definition, this is done by appending a new variable, x_0, at position 0, to the ordered tuple $(x_1, \ldots, x_n)$.

Depending on the operands f and g, this newly introduced variable can be of several types. Two kinds of variables will be of particular interest in this paper: useless and canalizing variables.

Definition 2 (Useless Variable)**.** *The variable x_0 of the Boolean function g operating on the ordered tuple $(x_0, x_1, \ldots, x_n)$, $n \geq 0$, is* useless *if and only if there exists a Boolean function f of arity n such that $g = f \star f$.*

Definition 3 (Canalizing Variable)**.** *The variable x_0 of the Boolean function g operating on the ordered tuple $(x_0, x_1, \ldots, x_n)$, $n \geq 0$, is* canalizing *if and only if there exists a Boolean function f of arity n such that one of the following cases occur:*

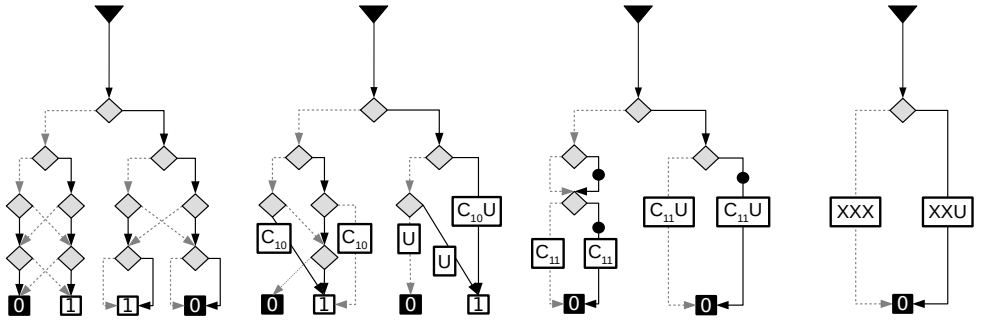
- $g = f \star 0$ *or* $g = 0 \star f$
- $g = f \star 1$ *or* $g = 1 \star f$*.*

For instance, in $g : (x, y) \mapsto x \wedge \neg y$, the variable x is canalizing. Indeed, let $f : y \mapsto \neg y$, then $g = 0 \star f$. Canalizing variables are dual to useless variables in the sense that their valuations could fix the output of the entire function. For the function g above, one has $g(0, y) = 0$ regardless of y.

A key observation for useless and canalizing variables alike is that the Shannon operator acts on *one* function f and produces a new function by appending a special variable to the ordered list of f. As such, for those special variables, this binary operator behaves more like a unary operator acting on Boolean functions. This simple observation is at the heart of the functional framework introduced next.

2 Ordered Functional Decision Diagrams

We introduce a new data structure, akin to ordered BDD, that we term Ordered Functional Decision Diagram or λDD−O.[1]



(a) λDD−S (b) λDD−O−UC$_{10}$ (c) λDD−O−NUC (d) λDD−O−NUCX

Figure 1: λDD−O graphs for $(x_0, x_1, x_2, x_3) \mapsto x_1 \oplus x_2 \oplus (\neg x_0 \wedge x_3)$.

2.1 Syntax and Semantics

Let Δ denote a finite set of letters, and Δ^* denote the set of all words (freely) generated by concatenating any finite number of letters from Δ. In particular, $\varepsilon \in \Delta^*$ denotes the empty word.

Definition 4 (λDD−O)**.** *A* λDD−O *(ϕ, n), $n \in \mathbb{N}$, is a parameterized recursive data structure defined as follows*

$$(\phi, 0) := (0, 0) \mid (1, 0)$$
$$(\phi, n+1) := \ell(\phi, n) \mid (\phi, n) \diamond (\phi, n),$$

[1]The relation between our work and the FDD of Kebschull et al. [10] is discussed in Section 4.1.

where ℓ is a letter in Δ. Observe that the letters in Δ as well as the binary operator '$\diamond$' increase the parameter n by exactly one. Notice also that the two operands of '$\diamond$' have necessarily the same parameter n.

A λDD-O can be represented as a directed acyclic graph, with (all) edges labeled with words in Δ^*. The representation requires three types of nodes: one root node ($\blacktriangledown$), two terminal nodes ($\square$ and $\blacksquare$), and a diamond node ($\diamond$). The structure is defined inductively as follows.

- a root node pointing to a terminal node;
- or a root node pointing to a λDD-O graph;
- or a root node pointing to a diamond node having two outgoing edges, each of which points to a λDD-O.

An edge can have only one label: composing $\xrightarrow{w_1}$ and $\xrightarrow{w_2}$ results in $\xrightarrow{w_1.w_2}$, where $w_1.w_2$ denotes the concatenation of w_1 and w_2.

The λDD-O data structure, or equivalently its graph representation, can be given, by induction, a semantics over Boolean functions. An elementary unary operator acts on a Boolean function f of arity $n \geq 0$ by appending a special variable to the input of f (which is an ordered tuple). For instance, the elementary operator δ_{u} defined by

$$\delta_{\mathsf{u}} : \mathbb{B}^{n\to 1} \to \mathbb{B}^{(n+1)\to 1}, \quad f \mapsto f \star f,$$

appends a useless variable to the input of f (seen as an ordered tuple). Similarly, each of the following elementary operators append a different canalizing variable to the input of f (Definition 3):

$$\delta_{\mathsf{c}_{0\,0}} : f \mapsto 0 \star f \qquad \delta_{\mathsf{c}_{1\,0}} : f \mapsto f \star 0$$
$$\delta_{\mathsf{c}_{0\,1}} : f \mapsto 1 \star f \qquad \delta_{\mathsf{c}_{1\,1}} : f \mapsto f \star 1$$

Definition 5 (Semantics of λDD-O)**.** *Let (ϕ, n) be a λDD-O graph.*

- $(\blacksquare, 0)$ *corresponds to the constant function 0 of arity zero;*
- $(\square, 0)$ *corresponds to the constant function 1 of arity zero;*
- *the letters in Δ correspond to elementary operators on Boolean functions;*
- *the concatenation of letters corresponds to the composition of operators;*
- *the empty word ε corresponds to the identity operator over Boolean functions;*
- *the operator '$\diamond$' corresponds to the Shannon operator defined over Boolean functions of the same arity (Definition 1).*
- *the parameter n corresponds to the arity of the Boolean function;*

A *model* of λDD-O is an instantiation of Δ with some letters. Semantically, Δ corresponds to a set of elementary operators, that we will also denote by Δ. For instance, the letters u and $\mathsf{c}_{1\,0}$ are used to encode the unary operators δ_{u} and $\delta_{\mathsf{c}_{1\,0}}$, respectively; we use similar letters for the remaining canalizing variables. The simplest possible model has no letters: Δ is empty and Δ^* contains one word of length zero, namely ε, which semantically corresponds to the identity operator $\delta_\varepsilon : f \mapsto f$. The λDD-O graphs obtained for $\Delta = \emptyset$, after merging isomorphic subgraphs, are known in the literature as Shannon Decision Diagrams, or SDD; we thus term this model λDD-S.

Example 1 (Running Example)**.** *The λDD−S graph* $(\phi, 4)$ *of the Boolean function*

$$(x_0, x_1, x_2, x_3) \mapsto x_1 \oplus x_2 \oplus (\neg x_0 \wedge x_3)$$

is depicted in Figure 1a where dashed and solid edges point respectively to the left and right operands of $\diamond$ *(recall that the operator* $\star$ *represented by '*$\diamond$*' is not commutative). For clarity, the terminal nodes are not merged and are explicitly labeled by their respective constant functions.*

The canonicity of the λDD−S data structure is obvious: each Boolean function has a unique λDD−S representation and every λDD−S represents unambiguously a unique Boolean function. In the next section, we detail how such canonicity is achieved for non-trivial models ($\Delta \neq \emptyset$).

2.2 Syntactic Reduction and Canonicity

Each letter in Δ comes with an introduction rule. Each introduction rule can be reversed to define its dual elimination rule. For instance, the introduction and elimination rules for the letter u are:

$$\texttt{intro-}\mathsf{u}\,\frac{(\phi, n) \diamond (\phi, n)}{\xrightarrow{\mathsf{u}} (\phi, n)}, \quad \texttt{elim-}\mathsf{u}\,\frac{\xrightarrow{\mathsf{u}} (\phi, n)}{(\phi, n) \diamond (\phi, n)} \ .$$

For convenience, we denote by $(\square, n)$ the graph

$$\underbrace{\xrightarrow{\mathsf{u}} \cdots \xrightarrow{\mathsf{u}}}_{n \text{ times}} (\square, 0)$$

and a similar shorthand notation will be used for the terminal node $\blacksquare$. The introduction rules for the letters c_{10} and c_{11} are as follows:

$$\texttt{intro-}\mathsf{c}_{10}\,\frac{(\phi, n) \diamond (\blacksquare, n)}{\xrightarrow{\mathsf{c}_{10}} (\phi, n)}, \qquad \texttt{intro-}\mathsf{c}_{11}\,\frac{(\phi, n) \diamond (\square, n)}{\xrightarrow{\mathsf{c}_{11}} (\phi, n)} \ .$$

The letters c_{00}, c_{01} are introduced similarly:

$$\texttt{intro-}\mathsf{c}_{00}\,\frac{(\blacksquare, n) \diamond (\phi, n)}{\xrightarrow{\mathsf{c}_{00}} (\phi, n)}, \qquad \texttt{intro-}\mathsf{c}_{01}\,\frac{(\square, n) \diamond (\phi, n)}{\xrightarrow{\mathsf{c}_{01}} (\phi, n)} \ .$$

We say that a graph is *reduced* if it is a fixed point for the introduction rules. Depending on Δ, the same graph may have distinct reduced representations. For instance, if Δ contains both c_{00} and c_{11}, then the graph $(\blacksquare, 0) \diamond (\square, 0)$ reduces to

$$\text{either} \quad \xrightarrow{\mathsf{c}_{00}} (\square, 0) \quad \text{or} \quad \xrightarrow{\mathsf{c}_{11}} (\blacksquare, 0) \ .$$

This observation can be formally captured as follows.

Definition 6 (Common Patterns)**.** *We say that a graph is a* common pattern *for the letters* $\ell, \ell' \in \Delta$ *if it can be reduced by their respective introduction rules.*

To guarantee canonicity, one needs to choose one particular representative for common patterns out of all the possible ones. To do so, we consider a *normalizing partial order* over the letters of Δ, defined as follows:

Definition 7 (Normalizing Partial Order)**.** *A partial order over* Δ *is* normalizing *if, whenever two letters share a common pattern, they must be ordered. Equivalently, incomparable letters do not share a common pattern.*

Fix a graph (ϕ, n). Let $>$ denote a normalizing partial order. It is easy to check that "sharing the common pattern (ϕ, n)" defines an equivalence relation over the letters.[2] The restriction of $>$ to such equivalence classes yields, by definition, a total order; its maximum letter is thus well-defined and unique. We denote it by $\text{LetterMax}_>(\phi, n)$. We can now normalize a reduced graph inductively over its structure:

```
let rec normalize (φ, n) =
  match (φ, n) with
  | (■, n') | (□, n') ⟶ (φ, n)
  | ℓ→ (φ', n') ⟶ (
    let φ'' = elim-ℓ (normalize (φ')) in
    let ℓ' = LetterMax_> (φ'') in
    intro-ℓ' (φ''))
  | (φ0, n') ⋄ (φ1, n') ⟶ (
    let φ' = (normalize (φ0)) ⋄ (normalize (φ1)) in
    let ℓ' = LetterMax_> (φ') in
    if ℓ' = ε then φ'
    else (intro-ℓ' (φ')))
```

In words, for a normalized (ϕ, n), normalizing $\xrightarrow{\ell} (\phi, n)$ amounts to (i) eliminating the letter ℓ (by reversing its introduction rule) then (ii) applying the introduction rule of the letter $\text{LetterMax}_>(\phi, n)$.

For normalized (ϕ_1, n) and (ϕ_2, n), normalize applies the introduction rule of corresponding to the letter $\text{LetterMax}_>\left((\phi_1, n) \diamond (\phi_2, n) \right)$.

A graph is said to be *normalized* if it is a fixed point for the procedure normalize. Observe that, by construction, a normalized graph is necessarily reduced: the procedure does not introduce diamond nodes. On the contrary, it may reduce more such nodes by normalizing their child graphs.

Lemma 1 (Uniqueness of Normalized Graphs)**.** *Let Δ be an λDD-O model equipped with a normalizing partial order $>$. Then, each Boolean function has a unique normalized graph with respect to $>$.*

Proof. Let f be a Boolean function. The proof is by induction on the arity of f. There are two Boolean functions of arity 0, namely the constant functions *0* and *1*, each of which has a unique normalized graph, namely $(\blacksquare, 0)$ for *0*, and $(\square, 0)$ for *1*. Suppose that the lemma holds for Boolean functions of arity n. Let f be of arity $n+1$.

Case 1. The normalized graph of f has the form $(\phi_1, n) \diamond (\phi_2, n)$. This means that both (ϕ_1, n) and (ϕ_2, n) are normalized. Suppose there exists another normalized graph for f of the form $\xrightarrow{\ell} (\phi, n)$. This means that $(\phi_1, n) \diamond (\phi_2, n)$ can be reduced, which contradicts the fact that it is normalized. Now suppose there exists another normalized graph for f of the form $(\phi'_1, n) \diamond (\phi'_2, n)$. By definition of the Shannon operator, (ϕ_1, n) and (ϕ'_1, n) are therefore semantically equivalent and are both normalized and of arity n. The induction hypothesis applies and implies that $(\phi_1, n) = (\phi'_1, n)$. The same holds for (ϕ_2, n) and (ϕ'_2, n), concluding this case.

Case 2. The normalized graph of f has the form $\xrightarrow{\ell} (\phi, n)$. This means that (ϕ, n) is normalized. Like the first case, there cannot be another normalized graph for f of the form $(\phi'_1, n) \diamond (\phi'_2, n)$. Suppose there exists another normalized graph for f of the form $\xrightarrow{\ell'} (\phi', n)$. Then ℓ and ℓ' are in the same equivalence class and, since both graphs are normalized, one gets $\ell < \ell'$ and $\ell > \ell'$. However, the partial order is total over equivalence classes, hence $\ell = \ell'$.

[2] However, "sharing a common pattern" is not an equivalence relation, as it lacks transitivity.

But then, (ϕ, n) and (ϕ', n) are two semantically equivalent normalized graphs of arity n. The induction hypothesis implies that they are necessarily the same, concluding the proof. □

A possible partial ordering over the letters we have introduced so far could be

$$\mathsf{u} > \{\mathsf{c}_{10}, \mathsf{c}_{11}\} > \{\mathsf{c}_{00}, \mathsf{c}_{01}\} \ .$$

Indeed, u shares a common, possibly different, pattern with each one of the other letters, whereas by definition c_{00} and c_{01} (and likewise c_{10} and c_{11}) do not share any common pattern and, therefore, do not need to be ordered. Normalizing $\xrightarrow{\mathsf{c}_{00}} (\square, 0)$ amounts to applying successively the two following rules:

$$\texttt{elim-}\mathsf{c}_{00}\ \frac{\xrightarrow{\mathsf{c}_{00}} (\phi, n)}{(\blacksquare, n) \diamond (\phi, n)}, \quad \texttt{intro-}\mathsf{c}_{11}\ \frac{(\phi, n) \diamond (\square, n)}{\xrightarrow{\mathsf{c}_{11}} (\phi, n)}$$

to obtain $\xrightarrow{\mathsf{c}_{11}} (\blacksquare, 0)$. The canonicity of λDD−O models can now be stated.

Theorem 1 (Canonicity of λDD−O representatives)**.** *Let Δ be a set of letters. Every Boolean function has a unique normalized λDD−O graph representation with respect to a given normalizing partial order over Δ.*

Proof. Any Boolean function has a unique λDD−S graph representation. The normalizing procedure with respect to $>$ is deterministic and terminates, as the diamond nodes of the λDD−S graph are finite. This ensures the existence of a normalized representative. Uniqueness is an immediate corollary of Lemma 1. □

The canonical representation of the Boolean function of Example 1 as a λDD−O graph for $\Delta = \{\mathsf{u}, \mathsf{c}_{10}\}$ with the normalizing partial order $\mathsf{u} > \mathsf{c}_{10}$ is shown in Figure 1b.

Remark 1. *Unlike BDD variants, the diamond nodes in λDD−O are not labeled with the integers denoting the indices (or positions) of variables. Such information can be fully retrieved from the length of the words labeling the edges and the nesting depth of diamond nodes.*

3 Negation Operator

We want to enrich our data structure with arity-preserving operators, such as negation. Arity-preserving operators are not elementary, since they do not increase the arity of their operands. This section details how such operators can be accounted for in λDD−O, without losing canonicity. We exemplify the main steps through the concrete example of the standard output negation operator on Boolean functions $\delta_\neg : \mathbb{B}^{n\to 1} \to \mathbb{B}^{n\to 1}$, defined by

$$(\delta_\neg f) : (x_0, \ldots, x_{n-1}) \mapsto \neg f(x_0, \ldots, x_{n-1}) \ .$$

Syntactically, the words labeling the edges will now include a new letter '•' that encodes the negation operator $\delta_\neg$.

The letter '•' is introduced by normalizing the representation of the square node $(\square, 0)$: [3]

$$\texttt{intro-}\bullet\ \frac{(\square, 0)}{\xrightarrow{\bullet} (\blacksquare, 0)} \ .$$

[3] Choosing $(\square, 0)$ over $(\blacksquare, 0)$ is arbitrary and has no impact on the treatment that follows.

Arity-preserving operators in general, and the negation operator in particular, interact with both the elementary operators and the Shannon operator. Those interactions have to be taken into account for the words labeling the edges of the graph to be canonical. There are two fundamental aspects of such interactions. The first one is the distributivity with the Shannon operator. The second one is the commutativity with elementary operators: when and how do arity-preserving operators commute with other operators?

The negation operator distributes over the Shannon operator, which makes it possible to expose it at the uppermost edge of the graph. Therefore, negating the data structure amounts to simply appending the letter '$\bullet$' to the uppermost word. The related normalization rule given below is very much similar to the `BDD` variants supporting negated edges.[4]

$$\texttt{norm-Shannon}\frac{(\xrightarrow{\bullet}(\phi,n))\diamond(\phi,n)}{\xrightarrow{\bullet}((\phi,n)\diamond(\xrightarrow{\bullet}(\phi,n)))}\ .$$

Commuting '$\bullet$' with the letters encoding the elementary operators allows to syntactically exploit the involution property of negation (i.e., the negation of a negated function is the function itself):

$$\texttt{norm-involution}\frac{\xrightarrow{\bullet}(\xrightarrow{\bullet}(\phi,n))}{(\phi,n)}\ .$$

However, commutation with elementary operators is not always possible. We give a (semantic) counter-example on Boolean functions, where syntactic commutation should not be allowed (the standard functional notations are given on the right and $\circ$ denotes the functional composition of operators).

$$\begin{aligned}(\delta_{\mathsf{c}_{1\,0}}\circ\delta_{\neg})0 &= 1\star 0 && (x\mapsto\neg x)\\ (\delta_{\neg}\circ\delta_{\mathsf{c}_{1\,0}})0 &= \neg(0\star 0) && (x\mapsto 1)\end{aligned}$$

To overcome this issue, we adopt in this work a weaker notion of commutativity.

Definition 8 (Quasi-commutativity)**.** *We say that an arity-preserving letter a* quasi-commutes *with an elementary letter ℓ if there exists an elementary letter ℓ_a (possibly different from ℓ) such that*

$$\ell.a = a.\ell_a\ .$$

(Observe that, as the notation suggests, the letter ℓ_a depends in general on a.) We will say that an alphabet Δ quasi-commutes *with a, or is $a-$stable, if $\ell\in\Delta$ implies $\ell_a\in\Delta$.*

Semantically, quasi-commutation of letters encodes the quasi-commutation of elementary operators and arity-preserving operators. For each arity-preserving operator a, the following normalization rules, defined for each elementary operator ℓ, exploit quasi-commutativity to expose arity-preserving operators before elementary operators.[5]

$$\texttt{norm-commutation-}(\ell,a)\frac{\xrightarrow{\ell.a}(\phi,n)}{\xrightarrow{a.\ell_a}(\phi,n)}\ .$$

Quasi-commutativity naturally extends to words: a word w quasi-commutes with an arity-preserving letter a if and only if all its letters quasi-commute with a. Normalizing partial orders

[4] Similarly to the choice of a terminal node, one may arbitrarily choose the dual reduction rule that normalizes the negation to the left edge instead.

[5] We stress that l must be an elementary letter, hence, one cannot apply `norm-commutation` on $\xrightarrow{\bullet.\bullet}(\phi,n)$. In such case, one should use `norm-involution` instead.

are extended to arity-preserving operators for the same reason they were introduced, that is, fixing a unique representative for common patterns. Thus, if ℓ_a is not unique for some arity-preserving operator a, then this also defines a common pattern and a normalizing partial order selects the maximum representative.

Notice that, thanks to the involution property of '$\bullet$', for any letter ℓ, $(\ell_\bullet)_\bullet = \ell$ (this is not necessarily true for other arity-preserving operators). Hence, one can make any alphabet Δ $\bullet$-stable by saturating it with the elements $\ell_\bullet$ for each $\ell \in \Delta$. Observe that the involution and commutation rules of the negation expose the $\bullet$ letter at the beginning of each word. Let us review some examples with respect to the useless and canalizing variables alphabet. The negation commutes with useless variables (i.e., $\mathsf{u}_\bullet = \mathsf{u}$); we are therefore allowed to rewrite the word '$\mathsf{u}.\bullet$' as '$\bullet.\mathsf{u}$'. However, it only quasi-commutes with all the other canalizing variables as follows:

$$\mathsf{c}_{00}.\bullet = \bullet.\mathsf{c}_{01} \qquad \mathsf{c}_{01}.\bullet = \bullet.\mathsf{c}_{00} \qquad (\mathsf{c}_{00_\bullet} = \mathsf{c}_{01})$$
$$\mathsf{c}_{10}.\bullet = \bullet.\mathsf{c}_{11} \qquad \mathsf{c}_{11}.\bullet = \bullet.\mathsf{c}_{10} \qquad (\mathsf{c}_{10_\bullet} = \mathsf{c}_{11})$$

Quasi-commutation explains the difficulty in adding (and normalizing) negation in some BDD variants like ZDD. Typically, a model that has the letter c_{01} without the letter $(\mathsf{c}_{01})_\bullet = \mathsf{c}_{00}$ cannot properly support the encoding and propagation of the negation over the data structure: the normalization becomes overly cumbersome while not offering any clear advantage.

In the presence of arity-preserving operators, the additional normalizing rules describing the interactions with the diamond node and the elementary operators are used during the normalizing process. A graph is therefore deemed normalized if and only if it is a fixed point for both the normalizing and the introduction rules with respect to the normalizing partial order extended to all operators, elementary and arity-preserving alike. The canonical representation of the Boolean function from Example 1 as a λDD−O graph for the alphabet $\Delta = \{\mathsf{u}, \mathsf{c}_{10}, \mathsf{c}_{11}\} \cup \{\bullet\}$,[6] with the normalizing partial order $\bullet > \mathsf{u} > \{\mathsf{c}_{10}, \mathsf{c}_{11}\}$, is given by Figure 1c.

Remark 2. *For the simple models we considered so far, the claims that the suggested partial orders are normalizing are relatively easy to check. However, the combinatorics becomes trickier as the number of operators grows. In Section 5, we present a more sophisticated model for which we rely on the Coq proof assistant to formalize and check canonicity.*

4 Functional Classification

The functional point of view suggests a natural way to classify and enumerate a vast class of ordered models. It turns out that a large body of already existing BDD variants can be seen, through the lenses of λDD, as special ordered models.

In this section, we exhaustively enumerate all arity-preserving and elementary operators that act on $f \in \mathbb{B}^{n\to 1}$ by *transforming its output*, that is, by transforming the Boolean $f(x_0, \ldots, x_{n-1})$. As a consequence, we introduce a new model, that is the most expressive possible in this class.

Remark 3. *Other arity-preserving and elementary transformations are possible and equally interesting. Let us cite, as an illustration, the Dual-Edge Based Variant [4], where one needs to add an arity-preserving variant δ_{DE} defined by:*

$$(\delta_{DE} f) : (x_0, \ldots, x_{n-1}) \mapsto \neg f(\neg x_0, \ldots, \neg x_{n-1}),$$

where the operator transforms the Boolean $f(\neg x_0, \ldots, \neg x_{n-1})$. Likewise, the Input Negation Invariant-Based Variant [2] uses two operators (one arity-preserving and one elementary) parameterized by a vector of Booleans $(a_0, \ldots, a_{n-1})$ acting on the Boolean $f(x_0 \oplus a_0, \ldots, x_{n-1} \oplus a_{n-1})$.

[6] Elementary and arity-preserving letters are kept separated on the writting of Δ for clarity.

The λDD framework can be instantiated to such variants by studying the possible interactions between these operators, similarly to what is done in the previous Sections. This is out of the scope of this paper, but planned in the near future.

An arity-preserving operator that acts on $f \in \mathbb{B}^{n\to 1}$ by transforming its output $f(x_0, \ldots, x_{n-1})$ necessarily has the form:

$$(\delta_p f) : (x_0, \ldots, x_{n-1}) \mapsto p(f(x_0, \ldots, x_{n-1}))$$

where p is a Boolean function in $\mathbb{B}^{1\to 1}$. There are $2^{2^1} (= 4)$ possibilities for p: the two constant functions *0* and *1* of arity 1, the identity function $\imath : x \mapsto x$, and the negation $\neg : x \mapsto \neg x$. When p is a constant function, δ_p is not injective and is therefore of little interest when it comes to canonical representations. When p is the identity $\imath$, then δ_p is the identity operator ε. Thus, one recovers the model λDD−S. Finally, when p is the negation, δ_p corresponds to $\delta_\neg$ and the so obtained model, termed λDD−S−N, enriches λDD−S with the (arity-preserving) negation operator.

In a similar fashion, we enumerate elementary operators that act on $f \in \mathbb{B}^{n\to 1}$ by combining its output $f(x_1, \ldots, x_n)$ with a fresh variable x_0. Such operator necessarily has the form

$$(\delta_p f) : (x_0, \ldots, x_n) \mapsto p(x_0, f(x_1, \ldots, x_n)) \ ,$$

where parameter p is now an element of $\mathbb{B}^{2\to 1}$. We can enumerate the $2^{2^2} (= 16)$ possibilities for p by combining two Boolean functions of arity 1 using the Shannon operator. All possible combinations are shown in the table below (p can be any function from the 16 cells of this table):

$(x, y) \mapsto 0$	$(x, y) \mapsto x$
$(x, y) \mapsto \neg x$	$(x, y) \mapsto 1$
$(x, y) \mapsto \neg x \wedge y$	$(x, y) \mapsto x \vee y$
$(x, y) \mapsto \neg(x \vee y)$	$(x, y) \mapsto \neg(\neg x \wedge y)$
$(x, y) \mapsto x \wedge y$	$(x, y) \mapsto \neg(\neg x \vee y)$
$(x, y) \mapsto \neg x \vee y$	$(x, y) \mapsto \neg(x \wedge y)$
$(x, y) \mapsto y$	$(x, y) \mapsto x \oplus y$
$(x, y) \mapsto \neg(x \oplus y)$	$(x, y) \mapsto \neg y$

We can in turn enumerate all possible elementary operators $\delta_p : \mathbb{B}^{n\to 1} \to \mathbb{B}^{(n+1)\to 1}$ (one for each p). We denote by $\pi_1 \in \mathbb{B}^{(n+1)\to 1}$ the projection operator returning the first input.

$f \mapsto 0$	$f \mapsto \pi_1$
$f \mapsto \delta_\neg(\pi_1)$	$f \mapsto 1$
$f \mapsto f \star 0$	$f \mapsto f \star 1$
$f \mapsto \delta_\neg(f \star 1)$	$f \mapsto \delta_\neg(f \star 0)$
$f \mapsto 0 \star f$	$f \mapsto \delta_\neg(1 \star f)$
$f \mapsto 1 \star f$	$f \mapsto \delta_\neg(0 \star f)$
$f \mapsto f \star f$	$f \mapsto f \star (\delta_\neg(f))$
$f \mapsto \delta_\neg(f \star (\delta_\neg(f)))$	$f \mapsto \delta_\neg(f \star f)$

Constant operators (2 cases), as well as the operators involving the projection π_1 (2 cases), are not injective. Therefore, they cannot be used for canonical representations. All operators in grayed cells can be fully captured by one of the elementary operators we have already introduced (related to useless or canalizing variables—see Definitions 2 and 3), possibly combined with the negation operator. The two remaining injective elementary operators reveal a new kind of variable which is neither useless nor canalizing.

Definition 9 (Xor Variable)**.** *Let δ_{x} denote the following elementary operator:*

$$\delta_{\mathsf{x}} : \mathbb{B}^{n\to 1} \to \mathbb{B}^{(n+1)\to 1}, \quad f \mapsto f \star \delta_{\neg}(f) \ .$$

A variable introduced with δ_{x} is called a xor variable*.*

Although one can define a model solely with δ_{x}, without supporting the negation as an extra operator, such a model would not necessarily be useful, as the reduction of the xor-variables cannot be performed in constant time over normalized subgraphs. Thus, such operator is much more relevant when the negation is properly supported (i.e., propagated and normalized as discussed in Section 3).

This enumeration suggests that a model defined using

$$\{\mathsf{u}, \mathsf{x}, \mathsf{c}_{00}, \mathsf{c}_{10}, \mathsf{c}_{01}, \mathsf{c}_{11}\} \cup \{\bullet\} \ ,$$

where the letter ' x' encodes δ_{x}, would be the most expressive, •-stable, model of the considered class as it has all the elementary operators plus the negation. The next section is entirely devoted to this model, termed λDD−O−NUCX. Table 1 summarizes some λDD−O models and their related variants. The TBDD [11] variant does not fit in ordered models as it uses a syntactic negation that does not correspond to the (functional) standard negation. [7] The last column of the table gives the alphabet of each model, split into two subsets: elementary letters (left) and arity-preserving letters (right).

It becomes apparent that Chain−BDD, Chain−ZDD [7] and ESRBDD−L_0 [8] are three possible implementations of the so-called λDD−O−UC_{10} model. The main difference between these variants resides in their respective, carefully designed, choices of encoding the involved elementary operators either as labels or special nodes. ESRBDD, for instance, encodes canalizing variables as nodes with one child. From a functional point of view, however, they are indistinguishable. Notice also that these three variants, as well as their related model, are not •-stable, thus, hindering the support of constant-time negation. This gives a clear insight into a fundamental limitation of these models.

The ordered models introduced so far actually form a complete lattice. Its (partial) Hasse diagram is depicted in Figure 2. The least upper bound (resp. greatest lower bound) of two models is the model induced by the union (resp. intersection) of their Δ alphabets. The first layer of the diagram has exactly 13 elements: one per elementary operator ($\{\mathsf{u}, \mathsf{x}, \mathsf{c}_{00}, \mathsf{c}_{01}, \mathsf{c}_{10}, \mathsf{c}_{11}\}$), one per negated elementary operator (not necessarily •-stable), and one involving the negation only. A model M_2 is more expressive than another model M_1 if and only if there is a path from M_1 to M_2 in the diagram. In particular, if there is no path between M_1 and M_2, then they are incomparable. This relation translates immediately to the number of nodes: the more expressive the model is, the smaller its number of nodes. Observe for instance, the graphs of Figure 1, where the number of nodes is decreasing from left to right.

4.1 Related Work

In this section, we outline fundamental differences between our work and, on the one hand, the classification of Darwiche [13] and, on the other hand, *Functional Decision Diagrams* of Kebschull et al. [10].

[7] However, this variant can be captured as a model enriched with a specific operator that captures the semantics of its syntactic negation.

Table 1: Existing BDD variants with their corresponding λDD ordered models.

Variant	**Model**	**Alphabet**
SDD	λDD-S	$\emptyset \cup \emptyset$
SDD+N	λDD-S-N	$\emptyset \cup \{\bullet\}$
ROBDD [1]	λDD-O-U	$\{\mathsf{u}\} \cup \emptyset$
ROBDD+N [12]	λDD-O-NU	$\{\mathsf{u}\} \cup \{\bullet\}$
ZDD [6]	λDD-O-C_{10}	$\{\mathsf{c}_{1\mathit{0}}\} \cup \emptyset$
Chain-BDD [7]	λDD-O-UC_{10}	$\{\mathsf{u}, \mathsf{c}_{1\mathit{0}}\} \cup \emptyset$
Chain-ZDD [7]	λDD-O-UC_{10}	$\{\mathsf{u}, \mathsf{c}_{1\mathit{0}}\} \cup \emptyset$
ESR-L_0 [8]	λDD-O-UC_{10}	$\{\mathsf{u}, \mathsf{c}_{1\mathit{0}}\} \cup \emptyset$
ESR [8]	λDD-O-UC_0	$\{\mathsf{u}, \mathsf{c}_{0\mathit{0}}, \mathsf{c}_{1\mathit{0}}\} \cup \emptyset$
DAGaml-O-NUCX	λDD-O-NUCX	$\{\mathsf{u}, \mathsf{x}, \mathsf{c}_{0\mathit{0}}, \mathsf{c}_{1\mathit{0}}, \mathsf{c}_{0\mathit{1}}, \mathsf{c}_{1\mathit{1}}\} \cup \{\bullet\}$

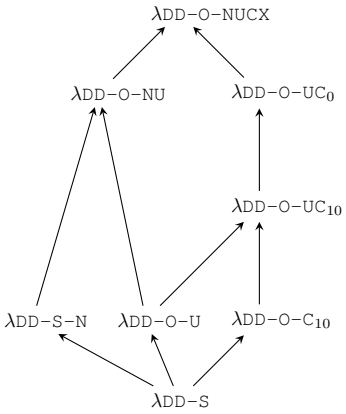


Figure 2: Some λDD ordered models organized in a lattice (cf. Table 1 for their corresponding variants).

Darwiche's Classification The λDD-based classification differs from Darwiche's work [13] that uses relative compactness and absolute worst-time complexity of standard queries to classify representations of Boolean functions.

Firstly, λDD is more fine grained. With respect to Darwiche's classification, many important variants like ROBDD [1], ROBDD+N [14], ZDD [6], Chain-DD [7], TBDD [11] and ESRBDD [8] are mostly indistinguishable from the vanilla variant SDD since they all fall in the same class. Indeed, all λDD-O models we presented in this work handle several queries (e.g., TAUTOLOGY, EQUIVALENCE, SAT, AnySAT, AllSAT, #SAT) in polytime, very much like ROBDD (cf. the time complexity discussion in Section 5.2).

Secondly, unlike Darwiche's classification, the λDD framework focuses primarily on getting more (functionally) expressive canonical models in a principled way.

Functional Decision Diagrams Functional Decision Diagrams (FDDs), introduced by Kebschull et al. [10], share some similarities with λDD, starting with their names. While, in both approaches, logic circuits are regarded (semantically) as Boolean functions, only λDD regards reduction rules as functors operating on Boolean functions. Furthermore, λDD relies entirely on the Shannon operator (or combinator) to deconstruct Boolean functions whereas FDD uses the positive Davio combinator. Recall that both combinators are *universal* and *elementary*: universal means that any Boolean function can be expressed as a combination of constant functions; elementary means that it operates on Boolean functions with the same arity by increasing the arity by exactly 1.

It is worth mentioning that Becker and Drechsler [15] identified a total of 12 distinct universal and elementary combinators having the same form as Shannon's, except that the branching relies on an arbitrary function p instead of the valuation of one (Boolean) variable. By allowing output negation, they further reduced this number to only 3, one of which is Shannon's; the other two combinators are the positive and negative Davio combinators. Recall that the (positive) Davio combinator is defined over Boolean functions of the same arity as $f \star_{\mathtt{D}^+} g$ as follows:

$$(f \star_{\mathtt{D}^+} g) : (x_0, x_1, \ldots, x_n) \mapsto f(x_1, \ldots, x_n) \oplus (x_0 \wedge g(x_1, \ldots, x_n)) \ .$$

Devising a data structure where the combinator is a parameter would be very relevant to compare and better understand the benefits and drawbacks of switching the underlying combinator. This would be a necessary first step towards a generic universal structure that allows even more complex combinators [16, 17, 18].

In fact, Shannon-based reduction rules can be transposed into *semantically equivalent* positive (or negative) Davio-based reduction rules. To better appreciate this, let us detail an example. To avoid any confusion, we use $\star_{\mathtt{S}}$ below to denote the Shannon operator. The following equation

$$f \star_{\mathtt{S}} f = f \star_{\mathtt{D}^+} 0$$

holds for any Boolean function f by definition of the combinators. Hence, a useless variable for a positive Davio-based λDD would be syntactically captured by the 'same' introduction rule of the letter $\mathsf{c}_{1\mathit{0}}$ we used for Shannon-based λDD, leading to the following sameness relation denoted by '$\leftrightsquigarrow$'.

$$\frac{(\phi, n) \diamond_{\mathtt{S}} (\blacksquare, n)}{\xrightarrow{\mathsf{c}_{1\mathit{0}}} (\phi, n)} \leftrightsquigarrow \frac{(\phi, n) \diamond_{\mathtt{D}^+} (\blacksquare, n)}{\xrightarrow{\mathsf{u}} (\phi, n)}$$

Table 2 summarizes this correspondence for the remaining elementary operators we have considered in this work. This observation shows the flexibility of the λDD framework and settles the first steps towards extending it to support Davio operators.

Table 2: Equivalence between Shannon-based reduction rules and Davio-based reduction rules

S	D^+	D^-
u	c_{10}	c_{10}
x	c_{11}	c_{11}
c_{00}	c_{00}	u
c_{01}	c_{01}	x
c_{10}	u	c_{00}
c_{11}	x	c_{01}

5 λDD−O−NUCX

We discuss in this section the most expressive canonical model, called λDD−O−NUCX, that supports all elementary operators of the class λDD−O, together with negation:

$$\Delta := \{\mathsf{u}, \mathsf{x}, \mathsf{c}_{00}, \mathsf{c}_{10}, \mathsf{c}_{01}, \mathsf{c}_{11}\} \cup \{\bullet\} \ .$$

We use the following introduction rule for the letter x (encoding xor-variables).

$$\texttt{intro-x}\frac{(\phi, n) \diamond \left(\xrightarrow{\bullet} (\phi, n)\right)}{\xrightarrow{\mathsf{x}} (\phi, n)} \ .$$

The letter 'x' commutes with '•', that is the word 'x.•' can be rewritten as '•.x'. Thus, Δ is •-stable.

Lemma 2. *The partial ordering* $\bullet > \{\mathsf{u}, \mathsf{x}\} > \{\mathsf{c}_{00}, \mathsf{c}_{10}, \mathsf{c}_{01}, \mathsf{c}_{11}\}$ *is normalizing.*

Proof. We detail the case of x and c_{00}: if these two letters share a common pattern, then there exist two graphs (ϕ, n) and (ϕ', n) such that

$$(\phi, n) \diamond \left(\xrightarrow{\bullet} (\phi, n)\right) = (\blacksquare, n) \diamond (\phi', n) \quad \text{(syntactic equality)}$$

Then, $(\phi, n) = (\blacksquare, n)$ and $(\phi', n) = \xrightarrow{\bullet} (\phi, n) = \xrightarrow{\bullet} (\blacksquare, n)$. So the two letters must be ordered, which is indeed the case. For this particular pattern, its normalized graph with respect to the chosen order is $\xrightarrow{\mathsf{x}} (\blacksquare, n)$. Likewise, one needs to check all common patterns for each pair of elementary operators. For the considered model, there is a total of $\binom{6}{2} = 15$ pairs that we also need to check. We formally checked all the pairs as part of the canonicity proof of the model using the Coq proof assistant (cf. the formal proof in the supplementary materials). □

The normalizing procedure described in Section 2 does not account for arity-preserving operators and their interactions with the elementary and Shannon operators. We thus need to extend it.

First, we need to prove that there exists a canonical representation for semantically equivalent words, like •.x.• and x. We do so by quasi-commuting the negation in order to expose it as a prefix; this is possible because Δ is •-stable. The normalization is then a consequence of the involution property of the negation (exploited via the rule `norm-involution`) and the fact that $\ell_\bullet$ is unique for each elementary $\ell \in \Delta$, and, therefore, $w_\bullet$ is also unique for each word formed by elementary letters.

Second, for the `norm-Shannon` rule, one needs to prove that there cannot be two distinct normalized graphs (ϕ_1, n), (ϕ_2, n) such that $\xrightarrow{\bullet} (\phi_1, n)$ and $\xrightarrow{\bullet} (\phi_2, n)$ are semantically equivalent.

If such graphs exist, then the interaction between the negation and the Shannon operator cannot be normalized. Indeed,

$$\left(\overset{\bullet}{\rightarrow}(\phi_2, n)\right) \diamond (\phi_2, n)$$

will be normalized using `norm-Shannon` to

$$\overset{\bullet}{\rightarrow}\left((\phi_2, n) \diamond \left(\overset{\bullet}{\rightarrow}(\phi_2, n)\right)\right)$$

which is semantically equivalent to (but distinct from) the graph

$$\left(\overset{\bullet}{\rightarrow}(\phi_1, n)\right) \diamond (\phi_2, n) \ .$$

Hence, we get two distinct graphs that are both normalized and semantically equivalent. This is an important observation if one wants to prove canonicity while using `norm-Shannon`. It highlights a subtle difficulty in properly handling negation in the presence of elementary operators that only quasi-commute with $\bullet$ (like $\mathsf{c}_{1\emptyset}$ in `ZDD`). As already stated, even the most recent canonical variants [7, 8] do not support negation. The functional point of view is crucial to grasp and overcome this difficulty.

With respect to the comments above, we prove a uniqueness lemma for λ`DD-O-NUCX` similar to Lemma 1, which is also performed by induction over the arity of Boolean functions. The induction step is, however, much more involved in this case. We formalized and checked the uniqueness claim in the Coq proof assistant. The details can be found in the companion formal proof. We also formally verified the instantiation of Theorem 1 to the λ`DD-O-NUCX` model. Hence, the following results holds.

Lemma 3 (Uniqueness of λ`DD-O-NUCX` Graphs)**.** *Every Boolean function has a unique normalized λDD-O-NUCX graph.*

Theorem 2 (Canonicity of λ`DD-O-NUCX`)**.** *For each Boolean function f there exists a unique normalized λDD-O-NUCX graph that is semantically equivalent to f.*

The λ`DD-O-NUCX` graph of the running example is given in Figure 1d; it only has one diamond node. Observe that, in addition to canonicity, the λ`DD-O-NUCX` graphs are *invariant* by negation, that is the graph of the function $\delta_{\neg} f$ only differs from the graph of f by a '$\bullet$' in the label of its uppermost edge. Thus, normalizing the negation of a λ`DD-O-NUCX` graph is performed in constant time.

5.1 Normalizing Logical Connectives

In practice, normalized λ`DD-O-NUCX` graphs are built by normalizing its logical connectives. We detail below the procedure '`andb`' that computes the conjunction of two normalized graphs. The algorithm is easily adaptable to the other operations. Computations are performed as usual, by recursively pushing the logical operator down to the child graphs of diamond nodes. For clarity, we denote a λ`DD` graph (ϕ, n) simply by ϕ, omitting arity n whenever unnecessary. We consider that the arity can be always extracted from ϕ by calling arity(ϕ).

```
let rec andb φ ψ =
  if φ = ψ then φ
  else if φ = •→ ψ then (■, arity(φ))
  else match φ with
    | (■, n) ⟶ (■, n)
    | •→ (■, n) ⟶ ψ
    | _ ⟶ (match ψ with
      | (■, n) ⟶ (■, n)
      | •→ (■, n) ⟶ φ
      | _ ⟶ (
        let φ0 = cofactor 0 φ
        and φ1 = cofactor 1 φ
        and ψ0 = cofactor 0 ψ
        and ψ1 = cofactor 1 ψ in
        shannon (andb φ0 ψ0)
                (andb φ1 ψ1) ))
```

'`cofactor` v_0 ψ' is defined below. Intuitively, it returns the graph of the Boolean function ψ when its first argument is set to v_0.

```
let cofactor v0 φ =
  match φ with
  | •→ φ' ⟶ •→(cofactorElem v0 φ')
  | _ ⟶ (cofactorElem v0 φ)

let cofactorElem v0 φ =
  match φ with
  | (■, n) ⟶ (■, n − 1)
  | l→ φ' ⟶ match l with
    | u ⟶ φ'
    | cbt ⟶ (if v0 = b then (t, n − 1) else φ')
    | x ⟶ (if v0 then •→ φ' else φ')
```

The '`shannon`' procedure forms a new graph as a diamond node of its argument; it relies on the '`push`' procedure, that normalizes the words labeling the edges.

```
let shannon φ ψ =
  if φ = ψ then push u φ
  else if φ = •→ ψ then push x φ
  else match ψ with
  | (■, n) ⟶ push c1ℓ φ
  | •→ (■, n) ⟶ push c1ı φ
  | _ ⟶ (match φ with
    | (■, n) ⟶ push c0ℓ ψ
    | •→ (■, n) ⟶ push c0ı ψ
    | •→ φ'0 ⟶ •→ (φ'0 ⋄ (•→ ψ))
    | _ ⟶ φ ⋄ ψ)
```

```
let push l φ =
  match φ with
  | •→ φ' ⟶ •→(pushElem l• φ')
  | _ ⟶ (pushElem l φ)

let pushElem l φ =
  match l, φ with
  | u, (■, n) ⟶ (■, n + 1)
  | c_0, (■, n) ⟶ (■, n + 1)
  | c_01, (■, n) ⟶ •→×→ (■, n)
  | c_11, (■, n) ⟶ ×→ (■, n)
  | _ ⟶ l→ φ
```

Time Complexity of `andb` Let $|\phi|$ denote the number of diamond nodes of the normalized graph (ϕ, n) of a Boolean function f, and let $|\phi|_{\mathtt{S}}$ denote the number of diamond nodes of the λDD-S graph of f. The algorithm `andb` can be applied almost identically to λDD-S graphs except for a minor edit to account for the terminal node $(\square, 0)$. The algorithm performs a simple structural induction on its inputs. Assuming memoization, the number of recursive calls is bounded by $\mathcal{O}(|\phi|_{\mathtt{S}} \times |\psi|_{\mathtt{S}})$. The time complexity of a single recursive call is $\mathcal{O}(1)$ as the complexity of `cofactor` is constant time (finite branching and no loops). Thus, the overall time complexity is $\mathcal{O}(|\phi|_{\mathtt{S}} \times |\psi|_{\mathtt{S}})$.

We have $|\phi|_{\mathtt{S}}$ is equal to $|\phi|$ plus the total size (or length) of all words in (ϕ, n) (each letter is a reduced diamond node). The size of any word in (ϕ, n) is bounded by n, the total number of variables. The total number of edges in (ϕ, n) is $1 + 2|\phi|$. Thus the total size of all words is bounded by $(1 + 2|\phi|)n$ and, therefore

$$|\phi|_{\mathtt{S}} \leq n + |\phi| + 2n|\phi| = \mathcal{O}(n \times |\phi|) \ .$$

This leads to an overall time complexity bounded by $\mathcal{O}(n^2 \times |\phi| \times |\psi|)$.

5.2 Time Complexity of Common Queries

In this section, we show that common polynomial queries on ROBDD (e.g., TAUTOLOGY, EQUIVALENCE, SAT, AnySAT, AllSAT, #SAT) are also polynomial on λDD-O-NUCX.

The simplest way to check for EQUIVALENCE((ϕ, n), (ψ, n)) is by hash-consing both ϕ and ψ, which has a linear time complexity in the size of both graphs leading to $\mathcal{O}(n \times (|\phi| + |\psi|))$.

For SAT, it suffices to check whether the graph is $(\blacksquare, n)$. In the worst case, the entire word (of size at most n) has to be checked, leading to a time complexity of $\mathcal{O}(n)$. Likewise for TAUTOLOGY.

One can compute #SAT inductively on the structure of a λDD-O graph:

```
let rec count (φ, n) =
  match (φ, n) with
  | (■, n) ⟶ 0
  | (□, n) ⟶ 2^n
  | •→ (φ', n') ⟶ 2^n − count(φ', n)
  | u→ (φ', n') ⟶ 2 × count(φ', n)
  | x→ (φ', n') ⟶ 2^n'
  | c0 0→ (φ', n') | c1 0→ (φ', n') ⟶ count(φ', n)
  | c0 1→ (φ', n') | c1 1→ (φ', n') ⟶ 2^n' + count(φ', n)
  | (φ0, n') ⋄ (φ1, n') ⟶ count(φ0, n') + count(φ1, n')
```

Using memoization, it can thus be computed in $\mathcal{O}(n \times |\phi|)$. Computing AnySat or AllSat can be performed similarly by induction over the structure. In particular, AnySat can be computed in $\mathcal{O}(n)$, and AllSat in $\mathcal{O}(n \times \#SAT(\phi))$.

6 Experimental Results

In this section, we assess the compression rates achieved by more and more expressive models in the light of the classification suggested by the λDD functional framework. Theoretically, it is clear that, the more expressive a model is, the lower the number of nodes will be. However, this comes at the cost of increasing the size of the words labeling the edges, in addition to the natural computational overhead. Therefore, it is unclear whether, in practice, the most expressive model supersedes all the others. Less expressive models might well be much more suitable for certain classes of functions.

To answer those questions, we selected two distinct sets of benchmarks that cover a reasonably wide range of Boolean functions. The first set consists of randomly generated CNF formulas (mostly sparse functions) from Satlib [19]. The second set consists of logic circuits (mostly dense functions) from `lgsynth91` [20] and `iscas99` [21].

To capture the size of a graph, we rely on three parameters: (1) the total number of (diamond) nodes, (2) the memory size of the words labeling the edges, and (3) an estimation of the overall size of the diagram, using the formula

$$\texttt{mem.} = 22\,\text{Bytes} \times \#\text{nodes} + \text{sizeof(labels)},$$

where sizeof(labels) is the accumulated size of all used labels. We used 22 Bytes per node, as this gives a reasonably good estimation of what it takes to encode a node as in [3].

We implemented all ordered models in OCaml as part of the `DAGaml` framework [22]. We used d4 [23] to preprocess all formulae into NNF in order to accelerate the compilation time as we are after all only interested in the structure of normalized graphs.

Remark 4. *We would like to stress the fact that the main focus of this section is not performance but rather the correlation between expressiveness of models and compression rates. We primarily seek to validate the theoretical intuitions and insights developed in the previous sections. While efficient implementation represents a challenging task, we believe that several important theoretical considerations have not received the attention they deserve. That being said, appendix A reports the runtime results for all our models and CUDD. We observed that CUDD is on average* 1.5× *faster. Indeed, our current implementation (in OCaml) lacks a garbage collector and does not support yet static and dynamic variable reordering. Notice that, in general, those features may have positive or negative effects on performance depending on the application.*

Table 3: Number of nodes, label size (in Bytes), and total estimated size (in Bytes) for successfully compiled benchmarks. Lowest total estimated sizes are highlighted in bold.

	DAGaml-O-U			DAGaml-O-NU			DAGaml-O-C$_{10}$			DAGaml-O-UC$_0$			DAGaml-O-NUCX		
	node	label	total	node	label	total	node	label	total	node	label	total	node	label	total
cm150a	131k	1.5M	4.1M	131k	1.5M	4.2M	131k	1.5M	4.1M	131k	1.3M	**3.9M**	131k	1.3M	4.2M
comp	589k	6.3M	18.1M	458k	5.2M	14.4M	426k	4.8M	13.3M	328k	3.9M	10.4M	197k	3.2M	**7.5M**
mux	131k	1.3M	4.0M	131k	1.4M	4.0M	131k	1.3M	4.0M	131k	1.1M	**3.7M**	131k	1.2M	4.0M
my_adder	720k	9.4M	23.8M	459k	5.9M	15.1M	803k	10.9M	26.9M	623k	7.5M	20.0M	262k	4.2M	**10.0M**
rot	624k	10.1M	22.5M	588k	9.6M	21.4M	2 424k	42.2M	90.7M	606k	9.2M	21.3M	565k	8.8M	**21.2M**
b04	25k	351k	860k	25k	358k	866k	122k	1 801k	4 235k	25k	311k	**818k**	25k	316k	873k
b07	30k	348k	962k	24k	278k	766k	41k	511k	1 337k	27k	293k	837k	21k	232k	**691k**
b09	13k	128k	402k	12k	120k	368k	15k	154k	460k	12k	106k	338k	9k	87k	**280k**
b11	11k	121k	345k	9k	100k	282k	15k	181k	491k	11k	108k	328k	9k	88k	**283k**
uf125-035	1 500	12k	45k	1 500	12k	45k	824	7k	25k	42	1.4k	**2.4k**	42	1.6k	2.5k
uf125-071	1 391	11k	42k	1 390	11k	42k	825	7k	25k	45	1.4k	**2.4k**	45	1.6k	2.6k
uf125-078	1 002	8k	30k	1 002	8k	30k	607	5k	18k	25	0.9k	**1.5k**	25	1.0k	1.6k
uf125-088	1 367	11k	42k	1 367	11k	42k	734	6k	22k	45	1.5k	**2.5k**	45	1.7k	2.7k
uf125-096	1 883	16k	57k	1 883	16k	57k	1 214	10k	37k	98	2.8k	**5.0k**	98	3.1k	5.3k
uf75-014	1 690	13k	50k	1 689	13k	50k	1 006	8k	30k	110	2.3k	**4.8k**	110	2.6k	5.0k
uf75-021	1 712	14k	51k	1 711	14k	51k	1 142	9k	35k	143	3.3k	**6.4k**	143	3.7k	6.8k
uf75-050	1 871	14k	55k	1 870	14k	56k	1 263	10k	38k	162	3.1k	**6.7k**	162	3.4k	7.0k
uf75-094	2 364	19k	71k	2 364	19k	71k	1 348	11k	41k	153	3.6k	**6.9k**	153	4.0k	7.3k
uf75-098	1 684	13k	50k	1 684	13k	50k	1 035	8k	31k	114	2.4k	**4.9k**	114	2.7k	5.2k

To highlight the difference between a model and its implementation, we will use the prefix DAGml: the implementation of a model λDD-O-XXX in DAGaml will be simply called DAGml-O-XXX. This distinction is important, as the same model may have rather different implementations. For instance, ESRBDD [8] and DAGml-O-UC$_0$ are two different implementations of the same model, namely λDD-O-UC$_0$. While we encode canalizing variables as letters labeling the edges, in [8] the same information is encoded as special nodes with one child.

The results of all successfully compiled benchmarks (with 6 hours timeout and 16GiB memory threshold) for all models are summarized in Table 3. For the other benchmarks, all models failed uniformly.

We can make several observations. First, some benchmarks seem to be beyond the scope of the refinement operated by the presented models. For instance, circuits cm150a and mux have similar total sizes ($\sim$ 4MB) across all models. A closer look shows that the number of nodes, as well as the size of their labels, are also nearly the same across all models. This means that the graphs of the different models are probably almost identical, as there are most likely very few canalizing and xor-variables.

The second observation is that, remarkably, in most examples, the strict decrease in the number of nodes comes with a decrease in the total size of labels. This means that, in practice, the potential increase in the size of labels is, by far, less important than the gain achieved by reducing the number of nodes.

Third, expressive models have much better compression rates compared to the canonical ROBDD. To get a clearer sense of this observation, we give in Table 4 the average compression ratios, relatively to λDD-O-U (which models ROBDD), of the number of nodes and the total memory estimation. One can see some expected patterns. For instance, λDD-O-C$_{10}$ (which models ZDD) is worse than ROBDD on dense functions but better on sparse functions. More importantly, the compression ratios attained by expressive models are striking for sparse functions, with a number of nodes divided by almost 25, while the gain is much less significant for dense functions. This is to be expected, as those functions do not necessarily exhibit the patterns captured by the models we presented. Indeed, a variable taken at random from the input of a dense function has little chance to be useless or canalizing.

Last but not least, the superiority of the most expressive model λDD-O-NUCX is not clearly established for sparse functions (CNF formulas). The model λDD-O-UC$_0$, albeit theoretically less expressive, seems to even slightly outperform λDD-O-NUCX in the sets of benchmarks we

Table 4: Average compression ratios relatively to DAGaml-O-U (ROBDD) for dense (lgsynth91 [20] and iscas99 [21]) and sparse (satlib [19]) functions.

	lgsynth91		iscas99	
	nodes	**mem.**	nodes	**mem.**
DAGaml-O-NU	1.14	1.13	1.12	1.11
DAGaml-O-C_{10}	0.49	0.47	0.49	0.48
DAGaml-O-UC_0	1.48	1.36	1.23	1.20
DAGaml-O-NUCX	1.57	1.31	1.57	1.48
	uf75-325		uf125-538	
	nodes	**mem.**	nodes	**mem.**
DAGaml-O-NU	1.00	1.00	1.00	1.00
DAGaml-O-C_{10}	1.75	1.74	1.76	1.74
DAGaml-O-UC_0	22.37	14.47	24.21	13.97
DAGaml-O-NUCX	22.38	13.63	24.22	13.08

Table 5: Average compression ratio relatively to DAGaml-O-U (ROBDD) for dual sparse functions.

	¬uf75-325		¬uf125-538	
	nodes	**mem.**	nodes	**mem.**
DAGaml-O-NU	1.00	1.00	1.00	1.00
DAGaml-O-C_{10}	0.96	0.96	0.96	0.95
DAGaml-O-UC_0	1.00	1.00	1.00	0.95
DAGaml-O-NUCX	22.38	13.63	24.22	13.08

considered. This can be explained by two factors: (i) accounting for more letters in the encoding of words has an overhead even if the additional letters are not used, and (ii) the support of the negation operator increases the size of the labels without necessarily reducing the number of nodes. This potential drawback should however, be mitigated by the fact that the slight increase caused by normalizing negation all over the structure allows to negate it in constant time, simply by editing its uppermost label. Depending on the targeted application, this advantage might be highly desirable.

Based on this last observation, it becomes clear that *dual sparse functions*, that is those functions that often evaluate to 1 instead of 0, are a suitable class for DAGaml-O-NUCX compared to λDD-O-UC_0 precisely because propagating the negation will exhibit the patterns the model is tailored for. To back up this intuition experimentally, Table 5 gives the same ratios as Table 4, but for the negations of the functions in the benchmarks of sparse functions. The gains achieved by the most expressive model are now salient.

Conclusion

The functional point of view developed in this paper helps getting a better understanding of how different existing variants of BDD are related, by abstracting away several implementation details in order to solely focus on how one constructs (or deconstructs) a Boolean function by adding (or removing) one variable at a time in a specific way. This approach allowed us to propose a new data structure with clear functional semantics, and to go beyond existing variants. We showed experimentally that the most expressive model achieves significant compression rates for sparse functions while being stable by negation. The resiliency of some dense functions to the operators we considered suggests that the scope of the patterns considered in this work could (and should) be widened.

References

[1] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, pp. 677–691, Aug. 1986.

[2] J. R. Burch and D. E. Long, "Efficient boolean function matching," ICCAD '92, (Los Alamitos, CA, USA), pp. 408–411, IEEE Computer Society Press, 1992.

[3] S. Minato, N. Ishiura, and S. Yajima, "Shared binary decision diagram with attributed edges for efficient boolean function manipulation," in *27th ACM/IEEE Design Automation Conference*, pp. 52–57, Jun 1990.

[4] D. M. Miller and R. Drechsler, "Dual edge operations in reduced ordered binary decision diagrams," in *Circuits and Systems, 1998. ISCAS '98. Proceedings of the 1998 IEEE International Symposium on*, vol. 6, pp. 159–162 vol.6, May 1998.

[5] A. Mishchenko, "An introduction to zero-suppressed binary decision diagrams," tech. rep., in 'Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, 2001.

[6] S. Minato, "Zero-suppressed bdds for set manipulation in combinatorial problems," DAC '93, (New York, NY, USA), pp. 272–277, ACM, 1993.

[7] R. E. Bryant, "Chain reduction for binary and zero-suppressed decision diagrams," *CoRR*, vol. abs/1710.06500, 2017.

[8] J. Babar, C. Jiang, G. Ciardo, and A. Miner, "Binary Decision Diagrams with Edge-Specified Reductions," in *Tools and Algorithms for the Construction and Analysis of Systems* (T. Vojnar and L. Zhang, eds.), Lecture Notes in Computer Science, pp. 303–318, Springer International Publishing, 2019.

[9] Q. He and M. Macauley, "Stratification and enumeration of boolean functions by canalizing depth," *CoRR*, vol. abs/1504.07591, 2015.

[10] U. Kebschull, E. Schubert, and W. Rosenstiel, "Multilevel logic synthesis based on functional decision diagrams," in *[1992] Proceedings The European Conference on Design Automation*, pp. 43–47, Mar. 1992. ISSN: null.

[11] T. van Dijk, R. Wille, and R. Meolic, "Tagged bdds: Combining reduction rules from different decision diagram types," in *FMCAD*, pp. 108–115, IEEE, 2017.

[12] F. Somenzi, "Binary decision diagrams," 1999.

[13] A. Darwiche and P. Marquis, "A Knowledge Compilation Map," *1*, vol. 17, pp. 229–264, Sept. 2002.

[14] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, (New York, NY, USA), pp. 40–45, ACM, 1990.

[15] B. Becker and R. Drechsler, "How many decomposition types do we need? [decision diagrams]," in *EDTC*, 1995.

[16] A. Bernasconi, V. Ciriani, G. Trucco, and T. Villa, "On decomposing boolean functions via extended cofactoring," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, (3001 Leuven, Belgium, Belgium), pp. 1464–1469, European Design and Automation Association, 2009.

[17] L. Amarú, P. Gaillardon, and G. D. Micheli, "Biconditional bdd: A novel canonical bdd for logic synthesis targeting xor-rich circuits," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1014–1017, March 2013.

[18] V. M. Bertacco, *Achieving Scalable Hardware Verification with Symbolic Simulation.* PhD thesis, Stanford, CA, USA, 2003. AAI3104197.

[19] H. H. Hoos and T. Stützle, "Satlib: An online resource for research on sat," pp. 283–292, IOS Press, 2000.

[20] S. Yang, "Logic synthesis and optimization benchmarks user guide: Version 3.0," tech. rep., MCNC Technical Report, Jan. 1991.

[21] F. Corno, M. Reorda, and G. Squillero, "Rt-level itc'99 benchmarks and first atpg results," *Design Test of Computers, IEEE*, vol. 17, pp. 44–53, Jul 2000.

[22] J. Thibault, "Abstract manipulations of directed acyclic graph using ocaml." `https://github.com/JoanThibault/DAGaml`, 2018.

[23] J.-M. Lagniez and P. Marquis, "An improved decision-DNNF compiler," IJCAI'17, pp. 667–673, AAAI Press.

[24] F. Somenzi, *Colorado University Decision Diagram package.* Colorado University.

A Run-time Comparison

Table 6 shows the computation time for `CUDD` [24] (implementing `ROBDD`) and `DAGaml` [22] compiling `lgsynth91` down to either `ROBDD` for `CUDD` or one of the implemented ordered model for `DAGaml`. All benchmarks were run on an server with an 8-core Intel Core i7 server with a clock rate of 2.7 GHz and 16GiB of RAM at 1.3GT/sec.

For `CUDD`, we used two configurations:

- `vr`: with variable reordering
- `no-vr`: without variable reordering

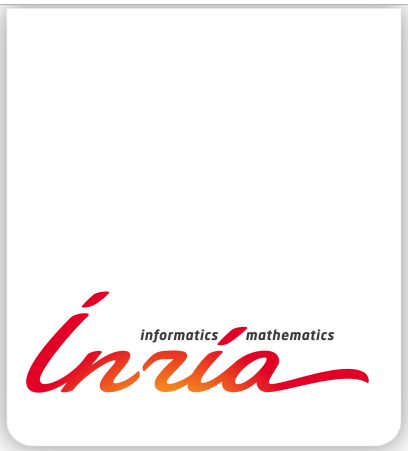
For `DAGaml`, we used the implementation of 4 different models:

- λ`DD-O-NU`, the model of `ROBDD`
- λ`DD-O-C`$_{10}$, the model of `ZDD`
- λ`DD-O-UC`$_0$, the model of `ESRBDD`
- λ`DD-O-NUCX`

On average, we observe that disabling variable reordering (`CUDD-vr` vs `CUDD-no-vr`) reduces by 45% Furthermore, all four `DAGaml` implementation of ordered models have similar performances. All shows a computation overhead of 45% These results tend to confirm the necessity to implement both garbage collectors and variable reordering, which are currently lacking to `DAGaml`. Again, we would like to stress the fact, that this paper's main focus is not on gaining performances but on the correlation between expressiveness of models and compression rates. While efficient implementation represents a challenging task, we believe that several theoretical problems related to the underlying models have not yet been addressed.

Table 6: Subset of the `lgsynth91` [20] experiment for which both `ROBDD/CUDD/ABC` (denoted `CUDD` in the table) and `DAGaml` succeessfully compiled in less than 6 hours and memory threshold of 16GiB. Each cell gives the compilation time (from Verilog) in seconds.

	CUDD		DAGaml			
	vr	no-vr	O-NU	O-$C_{1\theta}$	O-UC_{θ}	O-NUCX
9symml_orig	0.18	0.17	0.03	0.03	0.03	0.03
9symml_sweep	0.16	0.16	0.03	0.03	0.03	0.03
9symml_synth	0.17	0.17	0.03	0.03	0.03	0.03
C1355_orig	2.39	2.08	47.52	47.80	47.17	47.08
C1355_sweep	2.80	2.08	47.56	47.74	47.34	47.08
C1355_synth	1.79	19.08	31.88	32.57	31.71	31.49
C17_orig	0.16	0.16	0.01	0.01	0.01	0.01
C17_sweep	0.16	0.16	0.01	0.01	0.01	0.01
C17_synth	0.16	0.16	0.01	0.01	0.01	0.01
C1908_orig	1.23	2.38	6.09	6.31	6.18	6.05
C1908_sweep	6.90	14.74	14.11	14.44	14.13	14.02
C1908_synth	8.70	14.30	13.15	13.45	13.24	13.11
C432_orig	2.81	5.38	21.21	22.07	21.08	21.00
C432_sweep	2.86	11.86	119.98	120.96	118.88	119.02
C432_synth	2.84	11.14	58.44	60.01	58.04	58.17
C499_orig	2.38	3.26	32.94	33.40	32.62	32.52
C499_sweep	1.04	5.33	62.78	63.00	62.22	61.65
C499_synth	1.06	5.32	63.08	63.47	62.67	62.36
AVERAGE	-45%					



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399