Tailoring Binary Decision Diagram Compilation for Feature Models

Clemens Dubslaff[a,b], Nils Husung[c] and Nikolai Käfer[d]

[a] Formal System Analysis Cluster, Eindhoven University of Technology, Eindhoven, The Netherlands
[b] Centre for Tactile Internet with Human-in-the-Loop (CeTI), Dresden, Germany
[c] Dependable Systems and Software, Saarland University, Saarland Informatics Campus, Germany
[d] Institute of Theoretical Computer Science, Dresden University of Technology, Dresden, Germany

ABSTRACT

The compilation of feature models into *binary decision diagrams (BDDs)* is a major challenge in the area of configurable systems analysis. For many large-scale feature models such as the variants of the prominent Linux product line, BDDs could not yet be obtained due to exceeding state-of-the-art compilation capabilities. However, until now BDD compilation has been mainly considered on standard settings of existing BDD tools, barely exploiting advanced techniques or tuning parameters.

In this article, we conduct a comprehensive study on how to configure various techniques from the literature and thus improve compilation performance for feature models given in conjunctive normal form. Specifically, we evaluate preprocessing for *satisfiability solving (SAT)*, variable and clause ordering heuristics, as well as non-standard and multi-threaded BDD construction schemes. Our experiments on recent feature models demonstrate that BDD compilation of feature models greatly benefits from these techniques. We show that our methods enable BDD compilations of many large-scale feature models within seconds, including the whole ECOS feature model collection for which a compilation was previously infeasible.

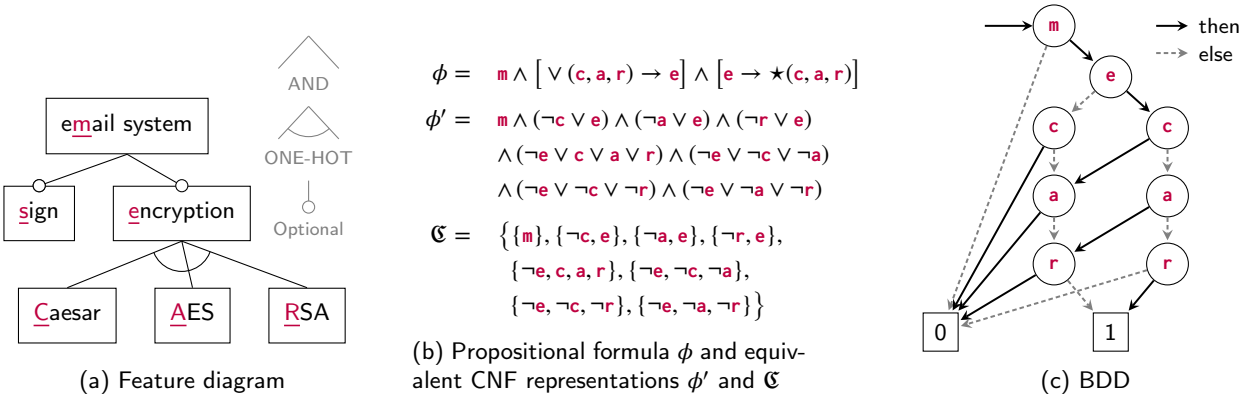
1. Introduction

In the feature-oriented modeling approach, features encapsulate optional or incremental functionalities of a software system. Each of the features can be configured to be included or excluded in the software, leading to *software product lines (SPLs)* as all *valid* configurations that can yield an actual software product [6, 60, 5]. Commonly, *feature models* such as *feature diagrams* are used to specify the set of valid configurations in an SPL [37]. Figure 1a shows an example of a feature diagram for a simple email SPL. The analysis of feature models is challenging, as the number of valid configurations is usually exponential in the number of features, and intricate side constraints need to be obeyed. Automated analysis of feature models is hence an active field of research [7, 23, 59, 28, 58]. These approaches exploit formal analyses of feature models based on their Boolean function semantics with abstract features modeled as Boolean variables: each variable is interpreted to be `true` if the feature is included in a configuration, and `false` otherwise. To benefit from recent advances in automated reasoning [9], the common approach is to translate feature models into propositional logic formulas in *conjunctive normal form* (CNF), i.e., conjunctions of clauses, which themselves are disjunctions of literals [7, 60]. CNFs constitute the standard input format for *satisfiability* (SAT) solvers, and many techniques for their manipulation and reasoning have been presented in the literature [9]. While reasoning tasks expressible through single SAT queries are known to scale well even on large feature models, more advanced tasks that involve counting the number of valid configurations through *model counting* (#SAT) or sophisticated feature model manipulation easily reach scalability boundaries [7, 58].

Knowledge compilation addresses these scalability challenges by transforming a model representation into another one, the latter admitting algorithmic advantages for specific tasks [14, 19]. For instance, compiling CNF into *deterministic decomposable negation normal form* (d-DNNF) enables to efficiently answer #SAT and hence uniform random sampling [18]. *Reduced ordered binary decision diagrams* (BDDs) [1, 13] constitute an even more versatile representation of Boolean functions than d-DNNFs, with many desirable properties such as efficient Boolean operators, SAT solving, #SAT computation, uniform random sampling, and—most distinctly—efficient equivalence checking [39]. BDDs are directed acyclic graphs where inner *decision nodes* are labeled by Boolean variables and have two outgoing edges. Following these edges specifies the assignment of the variable to `true` or `false`, respectively. Outcomes are modeled by two *terminal nodes* $\boxed{1}$ and $\boxed{0}$, standing for `true` and `false` evaluation, respectively. Reduction rules ensure that no redundant information by means of irrelevant decisions or duplicated isomorphic subdiagrams are contained in the diagram, leading to a concise representation of Boolean functions. Further, BDDs are ordered, i.e., the variables of decision nodes follow a given *variable order*. Reductions and ordering variables render BDDs canonical and enable efficient BDD operations and equivalence checking. Figure 1c shows an example BDD representing the valid configurations modeled in the feature diagram of Figure 1a.

The manifold benefits of BDDs as a data structure for Boolean functions come at a cost. Notably, their size can be exponential in the number of variables, such that the memory required to represent the BDD exceeds memory constraints.

*Authors in alphabetic order.
c.dubslaff@tue.nl (C. Dubslaff); husung@cs.uni-saarland.de (N. Husung); nikolai.kaefer@tu-dresden.de (N. Käfer)
ORCID(s): 0000-0001-5718-8276 (C. Dubslaff); 0009-0001-4375-3753 (N. Husung); 0000-0002-0645-1078 (N. Käfer)

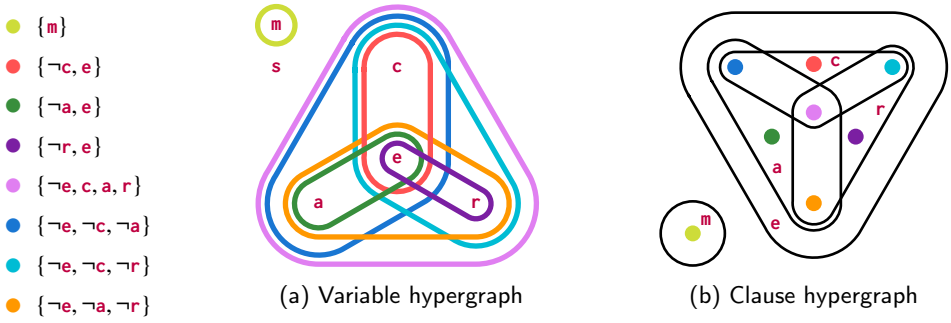


(a) Feature diagram

(b) Propositional formula ϕ and equivalent CNF representations ϕ' and $\mathfrak{C}$

(c) BDD

Figure 1: Three symbolic representations of a feature model for a simple configurable email system.



(a) Variable hypergraph

(b) Clause hypergraph

Figure 2: Variable and clause hypergraph for the CNF $\mathfrak{C}$ from Figure 1b.

The classical approach to construct BDDs is by a stepwise application of binary operations. Memory constraints then can be already exceeded during construction, even though the BDD representation for the targeted Boolean function would be small enough to fit into memory. This phenomenon is well-known as *peak-size explosion problem* and frequently arises when trying to construct BDDs for Boolean functions given in CNF [49, 22]. Typically, a BDD compilation from CNF formulas is performed by first constructing BDDs that represent the clauses of the CNF by disjunctive operations on literal BDDs, followed by joining them to a single BDD through applying conjunctive operations [23, 46, 16]. Such a stepwise *CNF-to-BDD compilation* corresponds to a bottom-up traversal of the abstract syntax tree of the CNF, assuming a left-deep bracketing of the clauses. Recent approaches also consider different bracketings, e.g., to yield balanced trees [46, 16, 48, 61, 21]. Besides following a bottom-up construction of BDDs from CNFs, top-down constructions [34] and combinations with SAT-based techniques to guide through CNFs have also been established [30].

The size of a BDD and thus the efficiency of operations is heavily influenced by the order of its variables, a well-known issue whose optimization involves to solve an NP-complete problem [11]. The variable-ordering problem is usually tackled by both static and dynamic variable-ordering heuristics. Static heuristics aim at establishing a good initial order [50] before starting the construction of the BDD. For CNFs, the standard heuristic is FORCE, a local search heuristic that moves dependent variables closer together [3]. MINCE is another prominent static heuristic that relies on recursively cutting hypergraphs generated from CNFs [4]. Dynamic heuristics reduce the size of the BDD during construction, most prominently through *sifting* [52, 23, 22]. For each variable, sifting tries every position in the current order and leaves it at the position with the smallest BDD size. While not as well-studied, variable-ordering heuristics applied on the dual problem to order clauses have shown to yield improvements in compilation performance and to avoid the peak-size explosion problem [4, 21].

Due to the superior position of BDDs in the knowledge compilation landscape [18], BDD compilation has raised much attention in the context of feature model analysis [44, 23, 59, 28, 58, 27, 21]. Existing work reports that under certain conditions, BDD compilation can scale well to feature models given as feature diagrams with up to 2,000 features and only few side-constraints [44]. However, real-world SPLs usually exhibit many side-constraints, which render dedicated techniques exploiting the hierarchical structure of feature diagrams far less effective [35]. For instance, when compiling the SPL corresponding to the 116 targets of the *embedded configurable operating system* (ECOS, v3.0) [8, 38],[1] memory or time bounds are easily exceeded

[1] The ECOS product lines are also referred to as the CDL collection in the literature, after the *component definition language* developed for ECOS.

even though these models contain only around 1,000 features [58]. For such models, advanced techniques are required to successfully construct their BDDs [27, 21]. More complex real-world feature models such as the well-known Linux kernel product line [54, 59] are still out of reach [28]. Towards advancing BDD compilation capabilities on real-world feature models, recent approaches focus on structural properties exhibited by such models that can be exploited. Notably, it is well known that real-world feature models contain many features in *ONE-HOT groups*, where in any valid configuration exactly one of the group's features is active. Using a specialized construction for such groups derived from feature diagrams, the tool LOGIC2BDD [23] enabled the construction of a real-world feature model with 17,365 features. There exist several methods to recover ONE-HOT groups from CNFs, predominantly employed as a preprocessing step to speed up SAT and #SAT solvers [29, 57]. For feature-model analysis, this idea of *ONE-HOT factorization* has recently been used to improve the FORCE variable-ordering heuristic [27]. Despite these advancements, literature on BDD compilation for feature-model analysis mainly focused on the comparison of existing tools and BDD engines in their standard configuration, only partly improving core aspects such as the variable ordering [28].

Our Approach. In this article, we investigate how various BDD construction techniques and configurations impact performance when compiling real-world feature models given in CNF. For this, we build upon and extend our work presented at the SPLC conference [21], where we conducted an initial comparison of existing BDD construction techniques. While most of the considered techniques were well known in the area of BDDs, they have barely been considered in combination and in the context of feature models [28, 48]. On a meta level, we hence analyze a product line itself (see Figure 6), where each configuration corresponds to a combination of techniques for CNF-to-BDD compilation. Specifically, we configure BDD compilation based on the following techniques:

(1) Equivalence-preserving preprocessing with PMC [41] or XOR and ONE-HOT[2] factorization [51].
(2) FORCE [3] or MINCE [4] variable and clause ordering heuristics.
(3) Left-deep or balanced CNF clause bracketing [48].
(4) Sequential or multi-threaded BDD construction [31].
(5) BDD compilers LOGIC2BDD [23] or OXIDD [31].

To the best of our knowledge, prior work on feature-model analysis has only considered sequential left-deep constructions with and without FORCE variable ordering and, in part, ONE-HOT factorization. We are the first to extensively evaluate those and other configurations on feature models. Moreover, we are not aware of any study assessing the impact of SAT preprocessing or FORCE clause ordering on CNF benchmarks.

To configure and analyze CNF-to-BDD compilation, we developed DIMAGIC [33], a tool that supports all aforementioned configurations. While the previous version of DIMAGIC [36, 21] was implemented in Python, we developed DIMAGIC 2.0 in Rust to enhance performance in feature model manipulations. This is particularly crucial for ReMINCE, our optimized variant of the MINCE heuristic, as well as our implementation of XOR and ONE-HOT factorization. ReMINCE mitigates construction errors found in the original MINCE binary [4] and uses the state-of-the-art hypergraph cutter KAHYPAR [53] for high-quality decomposition of CNF hypergraphs. As BDD engines, CUDD [56] serves as backend for the state-of-the-art feature model BDD compiler LOGIC2BDD [23] and OXIDD [31] provides the multi-threaded BDD engine with a dedicated CNF-to-BDD compilation interface.

We conduct experiments to evaluate BDD compilation configurations using a recent benchmark set of current feature models [26]. The number of all such configurations is exponential in the number of techniques, leading to almost 500 possible configurations even in the simplified setting we consider in our experiments. We hence adopt a structured approach towards finding configurations that perform best for feature models, also relying on the insights gained in previous experiments [21]. For this, we first analyze different preprocessing techniques, then ordering heuristics, and finally different construction schemes. After determining the best-performing configuration, we investigate its performance on the large feature models of the 116 ECOS product lines [8, 38] that were considered out of reach for CNF-to-BDD compilation in its entirety. Our tool DIMAGIC 2.0 and a replication package to reproduce our experiments are publicly available [33].

Our Contributions. To summarize, we conduct an extensive analysis on the performance of different CNF-to-BDD compilation configurations. We show that configuring the compilation has great impact on performance. In particular, the best performance within the configuration space is achieved by combining PMC preprocessing with ONE-HOT factorization, ReMINCE or MINCE variable and clause ordering, and multi-threaded construction following a balanced CNF bracketing with OXIDD. This configuration enables the compilation of each of the 116 ECOS feature models in less than 4 seconds on average.
Compared to the conference version [21], we contribute:

- DIMAGIC 2.0, a new implementation for configuring CNF-to-BDD compilation,
- ReMINCE, a reimplementation of MINCE that is open source, configurable, and mitigates construction errors of the original MINCE,
- advanced preprocessing with ONE-HOT and XOR factorization,
- theoretical considerations on BDD sizes depending on cutsets and hypergraph-based ordering heuristics,

[2] Here, we identify with XOR the binary operation that extends to the odd-parity function for higher arities, while we denote alternative group encodings by ONE-HOT, following the standard in circuit design. Note that XOR and ONE-HOT agree in the binary case and some literature uses both terms interchangeably.

- extended evaluation on a larger configuration space, including experiments on the performance of OxiDD and Logic2BDD with ONE-HOT factorization, and
- enhanced explanations, figures, and examples.

2. Foundations

In this section, we recall preliminaries and fix notations. The powerset of a set Y is denoted by 2^Y. We denote by $\mathbb{N}_{<i}$ the set of non-negative integers less than i, and by $|Y|$ the number of elements in Y, e.g., $|\mathbb{N}_{<i}| = i$. Further, $\mathbb{N}_{\leqslant i} := \mathbb{N}_{<i+1}$. A *total order* over Y is a bijection $\tau : \mathbb{N}_{<|Y|} \to Y$. Such an order can be seen as an ordered list $[\tau(0), \dots, \tau(|Y|-1)]$ where no element appears twice. Given total orders τ_Y and τ_Z over disjoint sets Y and Z, respectively, we define their concatenation as $\tau_Y +\!\!+ \tau_Z : \mathbb{N}_{<|Y|+|Z|} \to Y \cup Z$ where

$$(\tau_Y +\!\!+ \tau_Z)(i) := \begin{cases} \tau_Y(i) & \text{if } i < |Y| \\ \tau_Z(i - |Y|) & \text{otherwise.} \end{cases}$$

Configurations. Throughout this article, we assume that a configurable system over a fixed set of k features is given. The set of features is identified with a set of Boolean variables X of size $|X| = k$. A *configuration* is a mapping $\gamma : X \to \{0,1\}$, where $\gamma(x) = 0$ indicates that feature $x \in X$ is inactive in the configuration, while $\gamma(x) = 1$ stands for feature $x \in X$ being active. We denote the set of all possible configurations by $\mathsf{Conf}(X)$. A Boolean function over configurations is as a mapping $f : \mathsf{Conf}(X) \to \{0,1\}$. Constant Boolean functions $\mathbf{0}, \mathbf{1} : \mathsf{Conf}(X) \to \{0,1\}$ are defined by $\mathbf{0}(\gamma) := 0$ and $\mathbf{1}(\gamma) := 1$ for all $\gamma \in \mathsf{Conf}(X)$, respectively. A *partial configuration* $\delta : X \rightharpoonup \{0,1\}$ is a partial mapping defined only on the domain $\mathsf{Dom}(\delta) \subseteq X$ of features. Given a Boolean function $f : \mathsf{Conf}(X) \to \{0,1\}$, we write $f|_\delta$ for the Boolean function $f|_\delta : \mathsf{Conf}(X) \to \{0,1\}$ derived from f by setting features in $\mathsf{Dom}(\delta)$. Formally, $f|_\delta(\gamma) := f(\gamma_\delta)$ where $\gamma_\delta : X \to \{0,1\}$ and

$$\gamma_\delta(x) := \begin{cases} \delta(x) & \text{if } x \in \mathsf{Dom}(\delta) \\ \gamma(x) & \text{otherwise.} \end{cases}$$

Feature Models. Feature models specify *valid configurations* as a subset of $\mathsf{Conf}(X)$, where *feature diagrams* [37] constitute the most commonly used representation. Figure 1a depicts an example of a feature diagram specifying an email product line over features $X = \{\mathtt{m}, \mathtt{s}, \mathtt{e}, \mathtt{c}, \mathtt{a}, \mathtt{r}\}$. The semantics of a feature diagram is a Boolean function over configurations that is given by some propositional formula, e.g., ϕ in Figure 1b for the email system example. Essentially, this semantics interprets the hierarchical structure of the diagram and its decompositions: (1) the root is always active, (2) child features imply parent features, (3) parent features imply a child group, where AND groups require all their features to be enabled, OR groups require at least one, and ONE-HOT groups exactly one of their features, (4) features in an AND group may be marked as optional, and (5) arrows between features indicate additional implications as cross-tree constraints. In Figure 1a, the m feature has an AND group comprising optional features s and e, where the latter imposes a ONE-HOT group $\star(\mathtt{c}, \mathtt{a}, \mathtt{r})$.

Conjunctive Normal Forms. Towards a formal analysis of feature models, their propositional logic formula is usually translated into *conjunctive normal form* (CNF), i.e., into a formula of the form

$$\psi = \bigwedge_{i=0}^{m-1} \bigvee_{j=0}^{m_i-1} \ell_{i,j}$$

comprising $m \in \mathbb{N}$ *clauses*, where clause $i \in \mathbb{N}_{<m}$ consists of $m_i \in \mathbb{N}$ *literals* $\ell_{i,j} \in \{x, \neg x \mid x \in X\}$ for $j \in \mathbb{N}_{<m_i}$. Such CNF formulas ψ are conveniently given as a clause set $\mathfrak{C} = \{C_i \mid i \in \mathbb{N}_{<m}\}$ where $C_i = \{\ell_{i,j} \mid j \in \mathbb{N}_{<m_i}\}$ for $i \in \mathbb{N}_{<m}$. For instance, the formula ϕ for the feature model in Figure 1b is equivalent to the CNF ϕ', and ϕ' can be represented as the given set representation CNF $\mathfrak{C}$.

To ease notations, we define a negation operation $\sim$ on literals $\ell \in \{x, \neg x\}$ as

$$\sim\ell := \begin{cases} x & \text{if } \ell = \neg x \\ \neg x & \text{otherwise.} \end{cases}$$

Further, we write $\mathsf{Vars}(C)$ for the set of variables in a clause C, i.e., $\mathsf{Vars}(C) = \{x \in X \mid x \in C \text{ or } \neg x \in C\}$.

Formulas in *extended CNF* (XCNF) also allow the use of other clause operators than disjunctions "$\vee$", namely XOR "$\oplus$" and ONE-HOT "$\star$". Formally, an XCNF is given as a triple $\mathfrak{C} = \langle \mathfrak{C}_\vee, \mathfrak{C}_\oplus, \mathfrak{C}_\star \rangle$ of sets of clauses where we call any clause in $\mathfrak{C}_\vee$ a disjunctive clause, in $\mathfrak{C}_\oplus$ an XOR clause, and in $\mathfrak{C}_\star$ a ONE-HOT clause. We identify $\mathfrak{C}$ also with the set of all clauses $\mathfrak{C}_\vee \cup \mathfrak{C}_\oplus \cup \mathfrak{C}_\star$. For a configuration $\gamma \in \mathsf{Conf}(X)$ and variable $x \in X$, we set $\gamma(\neg x) = 1$ iff $\gamma(x) = 0$. The semantics of a clause C is given by Boolean functions $[\![C]\!]_\circ : \mathsf{Conf}(X) \to \{0,1\}$ where

- $[\![C]\!]_\vee(\gamma) = 1$ iff $\sum_{\ell \in C} \gamma(\ell) \geqslant 1$,
- $[\![C]\!]_\oplus(\gamma) = 1$ iff $\sum_{\ell \in C} \gamma(\ell)$ is odd, and
- $[\![C]\!]_\star(\gamma) = 1$ iff $\sum_{\ell \in C} \gamma(\ell) = 1$.

The semantics of an XCNF $\mathfrak{C}$ is defined as a Boolean function $[\![\mathfrak{C}]\!] : \mathsf{Conf}(X) \to \{0,1\}$ where $[\![\mathfrak{C}]\!](\gamma) = 1$ iff for all operators $\circ \in \{\vee, \oplus, \star\}$ and each $C \in \mathfrak{C}_\circ$ it holds that $[\![C]\!]_\circ(\gamma) = 1$. We say that a configuration γ *satisfies* a Boolean function f iff $f(\gamma) = 1$, and use $\mathsf{SAT}(f) := \{\gamma \in \mathsf{Conf}(X) \mid f(\gamma) = 1\}$ to denote the set of satisfying configurations. Similarly, we say that γ satisfies a clause $C \in \mathfrak{C}_\circ$ iff $[\![C]\!]_\circ(\gamma) = 1$ and an XCNF $\mathfrak{C}$ iff $[\![\mathfrak{C}]\!](\gamma) = 1$. A Boolean function, clause, or XCNF is *satisfiable* iff there exists a $\gamma \in \mathsf{Conf}(X)$ that satisfies it. Deciding whether $\mathfrak{C}$ is satisfiable is referred to as *SAT problem*. Counting models of $\mathfrak{C}$, i.e., computing $|\mathsf{SAT}([\![\mathfrak{C}]\!])|$, is referred to as *#SAT problem*. A clause $C \in \mathfrak{C}_\circ$ *subsumes* (or implies) a clause $C' \in \mathfrak{C}_\bullet$ iff $\mathsf{SAT}([\![C]\!]_\circ) \subseteq \mathsf{SAT}([\![C']\!]_\bullet)$. In particular, a ONE-HOT clause C subsumes an XOR clause C, which in turn subsumes a disjunctive clause C.

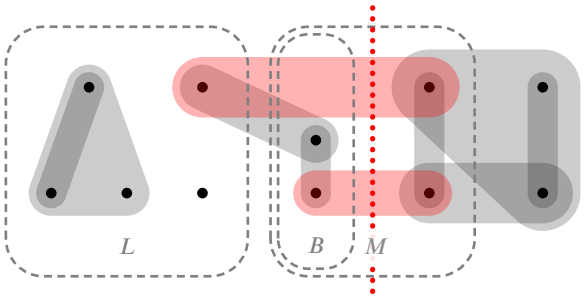


Figure 3: Example of a balanced mincut, assuming unit weights for all hyperedges. Cut hyperedges and the cutting separating B and $M \setminus B$ (dotted line) are highlighted in red.

Hypergraphs. A weighted *hypergraph* $\mathcal{H} = \langle V, E, w \rangle$ is a tuple comprising a set of *vertices* V, *hyperedges* $E \subseteq 2^V$, and *hyperedge weights* $w : E \to \mathbb{R}_+$. Given a set of hyperedges $D \subseteq E$, we denote their cumulative weight by $w(D) := \sum_{e \in D} w(e)$. Given an imbalance factor $\varepsilon \in (0, 1/2)$ and a set of vertices $M \subseteq V$, we call a set of vertices $B \subseteq M$ an *ε-balanced cut of M* if B and $M \setminus B$ have nearly equal size:

$$\max\left\{|B|, |M \setminus B|\right\} \leqslant (1+\varepsilon)\left\lceil \frac{|M|}{2} \right\rceil.$$

For a given a set of vertices $A \subseteq V$, we define $\mathsf{Cutset}_{\mathcal{H}}(A)$ as the set of edges in $\mathcal{H}$ cut by A, i.e.,

$$\mathsf{Cutset}_{\mathcal{H}}(A) := \left\{e \in E \mid e \cap A \neq \varnothing, e \cap (V \setminus A) \neq \varnothing\right\}.$$

The *cut weight* of A is the cumulative weight of edges in the cutset of A, i.e., $w\big(\mathsf{Cutset}_{\mathcal{H}}(A)\big)$. To describe cuts B of M with minimal cut weight, we extend the scope of the cut to the whole set of vertices. A *left-context of M* is a set of vertices $L \subseteq V \setminus M$. Given a left-context L, the *right-context of M* is defined as the set of vertices $R := V \setminus (M \cup L)$. Intuitively, left-context L and right-context R will be associated with a cut B and $M \setminus B$, respectively, but not be considered when determining cutsets. Then, we call B an *ε-balanced mincut of M in the left-context of L* if for all ε-balanced cuts B' of M we have

$$w\left(\mathsf{Cutset}_{\mathcal{H}}(L \cup B)\right) \leqslant w\left(\mathsf{Cutset}_{\mathcal{H}}(L \cup B')\right).$$

The setting of such a mincut in a left-context is illustrated in Figure 3. Computing such mincuts involves solving an NP-complete problem. To this end, many heuristics have been implemented to determine ε-balanced mincuts or approximations thereof, i.e., ε-balanced cuts that are nearly minimal [55]. We will assume such an implementation as an oracle where $\varepsilon\texttt{-BalancedMincut}_{\mathcal{H}}(L, M)$ returns a set of vertices B as above.

Let $\pi : \mathbb{N}_{<|V|} \to V$ be a total order on the hypergraph vertices, $i \in \mathbb{N}_{<|V|-1}$, and $V_i^\pi \subseteq V$ be the set of the first $i+1$ vertices in π, i.e., $V_i^\pi := \{\pi(j) \mid j \in \mathbb{N}_{<i+1}\}$. Then, $\mathsf{Cutset}_{\mathcal{H}}(V_i^\pi)$ is the i-th *cutset* of π in $\mathcal{H}$ and the *cutwidth* of π is defined $\max_i w\big(\mathsf{Cutset}_{\mathcal{H}}(V_i^\pi)\big)$. The *span* of a hyperedge $e \in E$ is $\max_{v \in e} \pi^{-1}(v) - \min_{v \in e} \pi^{-1}(v)$, i.e., the distance between the first and the last vertex $v \in e$ in the order π. The *total span* of π is the weighted sum of the spans for all hyperedges $e \in E$, or equivalently the weighted sum of the cutset sizes for π.

Given an XCNF $\mathfrak{C}$, we define the *variable hypergraph* $\mathcal{H}(\mathfrak{C}, b_\vee, b_\oplus, b_\star) := \langle X, \mathfrak{C}^*, w \rangle$ where vertices are variables and hyperedges $\mathfrak{C}^* = \left\{\mathsf{Vars}(C) \mid C \in \mathfrak{C}\right\}$ correspond to clauses disregarding the polarities of literals. The hyperedge weights are given by

$$w(e) := \sum_{\substack{C \in \mathfrak{C}_\vee \\ \mathsf{Vars}(C) = e}} b_\vee + \sum_{\substack{C \in \mathfrak{C}_\oplus \\ \mathsf{Vars}(C) = e}} b_\oplus + \sum_{\substack{C \in \mathfrak{C}_\star \\ \mathsf{Vars}(C) = e}} b_\star,$$

where $b_\vee$, $b_\oplus$, and $b_\star$ are called the *base weights* of the clause kinds. The *clause hypergraph* of $\mathfrak{C}$ is the hypergraph $\overline{\mathcal{H}}(\mathfrak{C}) := \langle \mathfrak{C}, \mathcal{E}, \bar{w} \rangle$ where each clause constitutes a vertex and each hyperedge contains the clauses with a common variable. Formally, let $\mathcal{E}_x := \{C \in \mathfrak{C} \mid x \in C \text{ or } \neg x \in C\}$ be the set of clauses mentioning variable $x \in X$. Then $\mathcal{E} := \{\mathcal{E}_x \mid x \in X\}$ and the hyperedge weights $\bar{w}(x) := |\mathcal{E}_x|$.

As an example, the variable and clause hypergraph for the CNF of the email product line are depicted in Figure 2. Each line corresponds to a hyperedge connecting the circled vertices, weights are elided. Assuming unit base weights, the cutwidth of the order `m c e a r` on vertices of the variable hypergraph is $\max\{0,4,6,4\} = 6$, the total span is $0{+}1{+}1{+}2{+}3{+}2{+}3{+}2 = 14 = 0{+}4{+}6{+}4$. For the order `m e c a r`, the cutwidth is $\max\{0,7,6,4\} = 7$ and the total span is $0{+}1{+}2{+}3{+}3{+}2{+}3{+}3 = 17 = 0{+}7{+}6{+}4$.

Binary Decision Diagrams. Let us assume a *variable order* on the set X as a total order $\pi : \mathbb{N}_{<k} \to X$. A *binary decision diagram* (BDD) is a labeled directed acyclic graph $\mathcal{D} = \langle N, \pi, \lambda, \mathsf{succ} \rangle$ over a finite set of inner nodes N and two distinct terminal nodes $\boxed{0}$ and $\boxed{1}$, a variable labeling function $\lambda : N \to X$, and a successor function $\mathsf{succ} : N \times \{0,1\} \to N \cup \left\{\boxed{0}, \boxed{1}\right\}$. For an inner node $n \in N$, the edge from n to $\mathsf{succ}(n,1)$ is called *then edge*, and the edge from n to $\mathsf{succ}(n,0)$ is called *else edge*. We assume BDDs to be ordered and reduced, i.e., for all inner nodes $n, n' \in N$ and $b \in \{0,1\}$:

(ordered) $n' = \mathsf{succ}(n,b)$ implies $\pi\big(\lambda(n)\big) < \pi\big(\lambda(n')\big)$

(reduced)

- $\mathsf{succ}(n,0) \neq \mathsf{succ}(n,1)$, and
- $\lambda(n) = \lambda(n')$ and $\mathsf{succ}(n,b) = \mathsf{succ}(n',b)$ imply $n = n'$.

By $|n|$ we denote the count of descendant nodes of n. Sometimes, we write π-BDD to denote a BDD with variable order π. Each node $n \in N \cup \left\{\boxed{0}, \boxed{1}\right\}$ in a BDD uniquely represents a Boolean function $[\![n]\!] : \mathsf{Conf}(X) \to \{0,1\}$, where for each $\gamma \in \mathsf{Conf}(X)$:

$$[\![n]\!](\gamma) := \begin{cases} b & \text{if } n = \boxed{b}, b \in \{0,1\} \\ [\![\mathsf{succ}\big(n, \gamma(\lambda(n))\big)]\!](\gamma) & \text{if } n \in N. \end{cases}$$

Figure 1c shows a BDD with an initial arrow to the node representing $[\![\mathfrak{C}]\!]$ of the email product line example CNF $\mathfrak{C}$

given in Figure 1b. As usual, *then* edges are depicted solid and *else* edges dashed. π-BDDs are canonical, i.e., for two π-BDDs $\mathcal{D}$ and $\mathcal{D}'$ over nodes N and N', respectively, and nodes $n \in N, n' \in N'$ with $[\![n]\!] = [\![n']\!]$, the subgraphs rooted at n and n' are isomorphic [24]. BDDs can be enhanced with *complement edges* to further reduce their size, employing the very same node n to represent both, a Boolean function $[\![n]\!]$ and its negation $\neg[\![n]\!]$ [1]. Technically, a complement bit is associated to each BDD edge, which indicates whether the function represented by the target node has to be complemented.

To practically use BDDs, we define notations for basic BDD building blocks and operations. For each operator of propositional logic, there is an algorithm applying that operator to BDD nodes, in particular `apply_and`, `apply_or`, `apply_xor`, and `apply_not` for the logic operators $\wedge$, $\oplus$, and $\neg$, respectively. For a variable $x \in X$, we denote by $\boxed{x}$ the BDD node that has a *then* edge to $\boxed{1}$ and *else* edge to $\boxed{0}$ and by $\boxed{\neg x}$ its negation ($\boxed{x}$). Therefore, BDDs can be used as a data structure for Boolean functions similar to propositional logic formulas. For instance, $\texttt{apply\_and}(n, n')$ and $\phi \wedge \phi'$ represent the same Boolean functions if $[\![n]\!] = \phi$ and $[\![n']\!] = \phi'$. The *apply* algorithms recursively decompose the operands at the top-most variable into their *then* and *else* branches and compose the results into a BDD node. Uniqueness, i.e., that no two distinct nodes represent the same Boolean function, is ensured on-the-fly during the latter composition step [12]. Applying a binary operator to two nodes n and n' yields a node with at most $|n| \cdot |n'|$ descendants. Using memoization, the time complexity is hence bounded by $\mathcal{O}(|n| \cdot |n'|)$. The operator `ite` implements a ternary if-then-else operation on BDDs, i.e., for any $\gamma \in \mathsf{Conf}(X)$ we have $[\![\texttt{ite}(n, t, e)]\!](\gamma) = 1$ iff $[\![n]\!](\gamma) = 1$ implies $[\![t]\!](\gamma) = 1$ and $[\![n]\!](\gamma) = 0$ implies $[\![e]\!](\gamma) = 1$.

3. CNF-to-BDD Compilation Techniques

Constructing BDDs from large-scale CNFs comes with challenging time and memory demands. Notably, not only the final BDD representing the CNF needs to fit into memory, but also the intermediate stages during construction. In fact, most approaches for CNF compilation suffer from the *peak-size explosion problem* where intermediate BDD sizes during construction exceed memory constraints while the resulting (canonical) BDD could well meet those constraints [34, 49, 22].

Standard Compilation Technique. The classical approach to compile a CNF $\mathfrak{C}_\vee$ into a π-BDD $\mathcal{D}$ with root node n where $[\![\mathfrak{C}_\vee]\!] = [\![n]\!]$ is done using a bottom-up approach that incrementally constructs the BDD by a left-deep execution of operands in the CNF [34, 23, 28, 16, 22, 31]. Algorithm 1 shows the pseudocode of this technique, which we extended for XCNF. Here, BDD nodes are first constructed for each clause (Lines 1–14), followed by BDDs for clauses successively being conjoined following the *clause order* ρ (Line 16). Note that the `apply` operations for clause construction (Lines 1–14) only take constant time when

Algorithm 1: Left-deep XCNF-to-BDD compilation

input : XCNF $\mathfrak{C} = \mathfrak{C}_\vee \cup \mathfrak{C}_\oplus \cup \mathfrak{C}_\star = \{C_i \mid i \in \mathbb{N}_{<m}\}$, variable order $\pi : \mathbb{N}_{<k} \to X$, and clause order $\rho : \mathbb{N}_{<m} \to \mathbb{N}_{<m}$
output: root node n in a π-BDD with $[\![n]\!] = [\![\mathfrak{C}]\!]$

1 **for** $i \coloneqq 0$ *to* $m-1$ **do**
2 $n_i \coloneqq \boxed{0}$
3 **if** $C_i \in \mathfrak{C}_\vee$ **then**
4 **for** $j \coloneqq 0$ *to* $m_i - 1$ **do**
5 $n_i \coloneqq \texttt{apply\_or}(n_i, \boxed{\ell_{i,j}})$
6 **else if** $C_i \in \mathfrak{C}_\oplus$ **then**
7 **for** $j \coloneqq 0$ *to* $m_i - 1$ **do**
8 $n_i \coloneqq \texttt{apply\_xor}(n_i, \boxed{\ell_{i,j}})$
9 **else if** $C_i \in \mathfrak{C}_\star$ **then**
10 $t \coloneqq \boxed{1}$; $e \coloneqq \boxed{0}$
11 **for** $j \coloneqq 0$ *to* $m_i - 2$ **do**
12 $e \coloneqq \texttt{ite}(\boxed{\ell_{i,j}}, t, e)$
13 $t \coloneqq \texttt{apply\_and}(t, \boxed{\sim\ell_{i,j}})$
14 $n_i \coloneqq \texttt{ite}(\boxed{\ell_{i,m_i-1}}, t, e)$
15 $n \coloneqq \boxed{1}$
16 **for** $i \coloneqq 0$ *to* $m-1$ **do** $n \coloneqq \texttt{apply\_and}(n, n_{\rho(i)})$
17 **return** n

literals are sorted descending according to π. The tool Logic2BDD [23] implements Algorithm 1 sequentially, while the construction integrated in OxiDD [31] also allows for concurrent `apply` and `ite` operators.

Advanced Techniques. There are many techniques presented in the literature to influence the performance of BDD construction. Most techniques alter the variable order π, since this has great impact on the size of the constructed BDD [11, 50]. Finding good variable orders involves to solve an NP-complete problem [11], such that static variable-ordering heuristics [50, 3, 4] or dynamic reordering [52] are the methods of choice. The clause order ρ is also known to impact construction speed and intermediate peak sizes [4, 48, 22]. Besides preprocessing the input CNF and orderings, also the compilation itself can be configured. Here, different construction schemes [48] or BDD engines for compilation [23, 25, 31] can be considered.

In this section, we describe various techniques to tune CNF-to-BDD compilation, focusing on existing approaches presented in the literature but also discussing new techniques in terms of CNF preprocessing and possible parallelization of the construction.

3.1. Preprocessing

Preprocessing the input CNF towards a CNF that exhibits more structure is well-established in the context of SAT solving, but has been barely considered for BDD compilation. We here focus on well-known techniques of CNF preprocessing and factorization.

3.1.1. CNF Preprocessing

CNF preprocessors from the literature transform a given CNF formula into another one to increase the performance for SAT or #SAT solvers [41, 40]. Most importantly in this context, these techniques only preserve satisfiability or the model count of the CNF, but not necessarily equivalence. In the context of CNF-to-BDD compilation, the goal is to yield a BDD representation of the very same Boolean function as encoded by the given CNF. Thus, we limit ourselves to well-known equivalence-preserving preprocessing techniques. The most basic such technique is *unit propagation* (UP, also *Boolean constraint propagation*, BCP [17]): For every unit clause $\{\ell\}$, it removes all other clauses containing ℓ, since these are satisfied iff $\{\ell\}$ is. Additionally, UP removes $\sim\ell$ from all clauses. UP can be implemented with linear worst-case time complexity.

The #SAT preprocessor PMC [41] implements three more advanced equivalence-preserving preprocessing techniques: backbone identification, occurrence reduction, and vivification. The backbone of a CNF $\mathfrak{C}$ is the set of literals that are true in every model, i.e., literals ℓ with $\gamma(\ell) = 1$ for all $\gamma \in \mathsf{Conf}(X)$ where $[\![\mathfrak{C}]\!](\gamma) = 1$. Each literal in the backbone can be added as a unit clause, and subsequent UP can reduce the CNF. Backbone identification is as hard as SAT, but structured approaches have shown to perform well in practice [45, 43]. Occurrence reduction is a simple technique that replaces clauses by some subsuming other clauses [41]. Using UP, vivification removes clauses and literals whose subclauses and negated literals are entailed by other clauses [47]. Occurrence reduction and vivification have a worst-case time complexity cubic in the CNF size [41]. In the following, we refer to "PMC" as the configuration that applies those three techniques.

3.1.2. ONE-HOT and XOR Factorization

While admitting a small BDD representation, some logical connectives such as ONE-HOT or XOR do not have a comparably small encoding in CNF. Any ONE-HOT clause $\star(\ell_0, \dots, \ell_{n-1})$ over n literals ℓ_i with different variables is equivalent to one disjunctive clause $\{\ell_i \mid i \in \mathbb{N}_{<n}\}$ and $n(n-1)/2$ disjunctive clauses $\{\sim\ell_i, \sim\ell_j\}$ with $i, j \in \mathbb{N}_{<n}$ and $i \neq j$. For instance, $\star(x, y, z)$ is equivalent to

$$(x \vee y \vee z) \wedge (\neg x \vee \neg y) \wedge (\neg x \vee \neg z) \wedge (\neg y \vee \neg z).$$

A (non-empty) XOR clause $\oplus(\ell_0, \dots, \ell_{n-1})$ generally requires even 2^{n-1} disjunctive clauses $L \cup \{\sim\ell \mid \ell \in L'\}$, where $L, L' \subseteq \{\ell_i \mid i \in \mathbb{N}_{<n}\}$, $L \cap L' = \varnothing$, and $|L'|$ is even. For instance, $\oplus(x, y, z)$ is equivalent to

$$(x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z).$$

At the same time, ONE-HOT and XOR clauses of size n can be represented using BDDs with only $2n - 1$ inner nodes. Hence, BDD compilation can greatly benefit from directly compiling ONE-HOT and XOR clauses (see Algorithm 1) instead of compiling their disjunctive clause counterparts. However, most (feature) models are given as plain CNFs, not admitting direct access to ONE-HOT and XOR clauses.

Algorithm 2: ONE-HOT factorization

input : set of disjunctive clauses $\mathfrak{C}_\vee$
output: set of ONE-HOT clauses $\mathfrak{C}_\star$,
set of subsumed disjunctive clauses $\mathcal{S}_\star$
time : $\mathcal{O}(\sum_{C \in \mathfrak{C}_\vee} |C|^2)$

1 $\mathfrak{C}_\star := \varnothing$; $\mathcal{S}_\star := \varnothing$
2 **foreach** $C \in \mathfrak{C}_\vee$ *with* $|C| \geqslant 3$ **do**
3 **if** $\{\{\sim\ell, \sim\ell'\} \mid \ell, \ell' \in C, \ell \neq \ell'\} \subseteq \mathfrak{C}_\vee$ **then**
4 $\mathfrak{C}_\star := \mathfrak{C}_\star \cup \{C\}$
5 $\mathcal{S}_\star := \mathcal{S}_\star \cup \{\{\sim\ell, \sim\ell'\} \mid \ell, \ell' \in C, \ell \neq \ell'\}$
6 **return** $\mathfrak{C}_\star, \mathcal{S}_\star$

Under the assumption that the given CNF resulted from the above standard transformations applied on ONE-HOT or XOR groups, techniques to recover such groups showed great success in the context of SAT and #SAT solvers [29, 57]. This motivates to investigate CNF-to-BDD compilation where ONE-HOT and XOR clauses are recovered and factored out into an XCNF, which is subsequently compiled into a BDD using Algorithm 1. Throughout this section, we assume an input CNF $\mathfrak{C}_\vee$ that is *simplified*, i.e., there are no clauses with both positive and negative occurrences of the same variable, and unit propagation has been applied exhaustively. The latter avoids a special case during ONE-HOT recovery, namely to look for unit clauses subsuming a required two-literal clause.

ONE-HOT Recovery. Algorithm 2 can detect ONE-HOT groups with at least three literals in disjunctive clauses $\mathfrak{C}_\vee$, returning ONE-HOT clauses $\mathfrak{C}_\star$ with their subsumed set of original disjunctive clauses $\mathcal{S}_\star$. This leads to the factorized XCNF $\mathfrak{C} = \langle \mathfrak{C}_\vee \setminus \mathcal{S}_\star, \varnothing, \mathfrak{C}_\star \rangle$ where $[\![\mathfrak{C}_\vee]\!] = [\![\mathfrak{C}]\!]$ and which thus can be used for BDD compilation. Note that if we made the algorithm recover two-literal ONE-HOT clauses by lowering the bound in Line 2, it would add both $\{\ell, \ell'\}$ and $\{\sim\ell, \sim\ell'\}$ to $\mathfrak{C}_\star$. We cover this case in the subsequent XOR recovery. Algorithm 2 has a running time linear in the number of clauses and quadratic in the clause size.

XOR Recovery. Heule [29] presented a simple algorithm for XOR recovery by sorting the disjunctive clauses for efficient processing. First, each clause is sorted by its variables, followed by sorting the clause set (1) lexicographically by their variables and (2) by the parity of the negated literal count. This ensures that the disjunctive clauses forming an XOR clause are next to each other. The final XOR recovery is then just a linear scan over the clauses.

However, it is quite common that for an XOR group of size n, some of the corresponding 2^{n-1} disjunctive clauses of size n are not present in the clause set, but instead smaller clauses subsuming them [57]. Algorithm 3 is more costly than the simple algorithm but also recovers the XOR clause in such a case. Its worst-case running time is quadratic in the number of clauses and exponential in the clause size. Therefore, large clauses incur high computational costs. At the same time, it is rather unlikely to actually recover a large

Algorithm 3: XOR factorization

input : set of disjunctive clauses $\mathfrak{C}_\vee$
output: set of XOR clauses $\mathfrak{C}_\oplus$,
set of subsumed disjunctive clauses $\mathcal{S}_\oplus$
time : $\mathcal{O}\big(|\mathfrak{C}_\vee| \sum_{C\in\mathfrak{C}_\vee} 2^{|C|}\big)$

$\mathfrak{C}_\oplus \coloneqq \varnothing$; $\mathcal{S}_\oplus \coloneqq \varnothing$; $\mathcal{D} \coloneqq \varnothing$
foreach $C \in \mathfrak{C}_\vee$ *with* $|C| \geqslant 2$ **do**
  $\mathcal{D} \coloneqq \mathcal{D} \cup \{C\}$ // "done" clauses
  $\mathcal{F} \coloneqq \{C\}$ // "found" clauses of length $|C|$
  foreach $C' \in \mathfrak{C}_\vee \backslash \mathcal{D}$ *with* $\mathsf{Vars}(C') \subseteq \mathsf{Vars}(C)$ **do**
    $V \coloneqq \mathsf{Vars}(C \backslash C')$
    foreach $P \subseteq V$ **do** // expand C' to length $|C|$
      $C'' \coloneqq C' \cup P \cup \{\neg x \mid x \in V \backslash P\}$
      if $|\{\ell \in C'' \mid \sim\ell \in C\}|$ *is even* **then**
        $\mathcal{F} \coloneqq \mathcal{F} \cup \{C''\}$
  if $|\mathcal{F}| = 2^{|C|-1}$ **then**
    $\mathfrak{C}_\oplus \coloneqq \mathfrak{C}_\oplus \cup \{C\}$
    $\mathcal{S}_\oplus \coloneqq \mathcal{S}_\oplus \cup (\mathcal{F} \cap \mathfrak{C}_\vee)$
return $\mathfrak{C}_\oplus, \mathcal{S}_\oplus$

XOR. Thus, one usually imposes an upper bound on the literal count of a potential XOR clause in practice.

Combined Recovery. When using XOR factorization in isolation, one would consider the XCNF $\mathfrak{C} = \langle \mathfrak{C}_\vee \backslash \mathcal{S}_\oplus, \mathfrak{C}_\oplus, \varnothing \rangle$ as input of Algorithm 1 for BDD construction. When combining ONE-HOT and XOR factorization, ONE-HOT recovery is applied first. Assuming that $\mathfrak{C}_\vee$ does not contain redundant clauses, XOR recovery will not find an XOR clause $\oplus(\ell_0, \dots, \ell_{n-1})$ in $\mathfrak{C}_\vee$ if ONE-HOT recovery identifies $\star(\ell_0, \dots, \ell_{n-1})$ in $\mathfrak{C}_\vee$, since the ONE-HOT clause subsumes the XOR clause. So we use $\mathfrak{C}_\vee \backslash \mathfrak{C}_\star$ as input of XOR factorization, since the clauses in $\mathcal{S}_\star$ may still be necessary to recover some XORs. For BDD construction, the XCNF $\mathfrak{C} = \langle \mathfrak{C}_\vee \backslash (\mathcal{S}_\star \cup \mathcal{S}_\oplus), \mathfrak{C}_\oplus, \mathfrak{C}_\star \rangle$ is then used as input for Algorithm 1.

3.2. Ordering Heuristics

The size of a BDD crucially depends on its variable order [13, 11], and many heuristics have been presented in the literature to establish good variable orders for BDDs compiled from circuits or CNFs [50]. Dually, construction performance and the peak size of a BDD depend on the order of gates processed in circuits and the clause order in CNFs, respectively (see Algorithm 1). Most heuristics follow the well-known principle that good orders place dependent variables and clauses close to each other. These dependencies are captured in the variable or clause hypergraph, respectively. A good quality measure on the close placement of dependent variables is then provided by the total span of an order on hypergraph vertices, i.e., the accumulated maximal distances of directly connected vertices in the order.

3.2.1. Theoretical Considerations

Variable Ordering. The intuition behind minimizing the total span for good variable orders can be explained as follows: the lower the total span, the closer the variables that have common clauses are placed in the order. Between such close variables, information about a variable decision has to be carried over to fewer other variable decisions before ultimately deciding whether a clause is satisfied or not. The textbook example for extremal cases of this phenomenon is the CNF

$$\varphi_m = (x_0 \vee y_0) \wedge (x_1 \vee y_1) \wedge \dots \wedge (x_m \vee y_m)$$

that has a variable hypergraph with $m+1$ isolated pairs of variables, each connected by one edge. Choosing the variable order

$$[x_0, \dots, x_m, y_0, \dots, y_m]$$

yields a BDD with 2^{m+2} nodes, while the interleaved variable order

$$[x_0, y_0, \dots, x_m, y_m]$$

yields a BDD with only $2\cdot(m+2)$ nodes. The latter order puts variables in a common clause next to each other, minimizing the total span. The role of the variable order can be captured by an upper bound on the BDD size:

Theorem 1. *Let $\mathfrak{C}$ be an XCNF over a set of variables X, $\mathcal{H} = \mathcal{H}(\mathfrak{C}, 1, 1, 1) = \langle \mathfrak{C}, \mathcal{E}, w \rangle$ its variable hypergraph with unit base weights, and $\pi : \mathbb{N}_{<k} \to X$ a variable order. Then*

$$3 + \sum_{i=0}^{k-2} 2^{w\left(\mathsf{Cutset}_{\mathcal{H}}(X_i^\pi)\right)}$$

is an upper bound for the node count of the π-BDD for $\mathfrak{C}$.

Recall that $X_i^\pi = \{\pi(j) \mid j \in \mathbb{N}_{<i+1}\}$. A similar upper bound has been devised by Huang and Darwiche for CNFs and relying on the cutwidth, i.e., the maximum cutset size [30]. Our theorem provides a generalization from CNF to XCNF and typically yields a tighter bound by taking the individual cutset sizes into account.

Proof. Each node n in a π-BDD corresponds to one or multiple partial configurations $\delta : X \rightharpoonup \{0,1\}$ whose domains are the set of variables that occur in π before n's associated variable $\lambda(n)$. In general, there are 2^i partial configurations δ with $\mathsf{Dom}(\delta) = \{\pi(j) \mid j \in \mathbb{N}_{<i}\}$. However, $[\![\mathfrak{C}]\!]|_\delta$ may be the same Boolean function for some of such δ and therefore be represented by the same BDD node. We now use cutsets to establish an upper bound on the number of different functions $[\![\mathfrak{C}]\!]|_\delta$, which immediately translates to an upper bound on the BDD node count.

If $\mathsf{Dom}(\delta) = X$ for a (partial) configuration δ, then $[\![\mathfrak{C}]\!]|_\delta$ is either $\mathbf{0}$ or $\mathbf{1}$. Considering the partial configuration $\delta_\varnothing$ with $\mathsf{Dom}(\delta_\varnothing) = \varnothing$, the function $[\![\mathfrak{C}]\!]|_{\delta_\varnothing}$ is just $[\![\mathfrak{C}]\!]$. These functions correspond to the 3 in the upper bound sum. Considering $i \in \mathbb{N}_{\leqslant k-2}$, it hence suffices to show that

$$\left|\left\{[\![\mathfrak{C}]\!]|_\delta \mid \mathsf{Dom}(\delta) = X_i^\pi\right\} \setminus \{\mathbf{0}\}\right| \leq 2^{w\left(\mathsf{Cutset}_{\mathcal{H}}(X_i^\pi)\right)}.$$

To this end, we consider each clause $C \in \mathfrak{C}$ separately and will show that unless $[\![C]\!]_\circ|_\delta = \mathbf{0}$, it may be (1) either

of two functions if $C \in \mathsf{Cutset}_{\mathcal{H}}(X_i^\pi)$, or (2) just a single function if $C \notin \mathsf{Cutset}_{\mathcal{H}}(X_i^\pi)$. Here, $\circ$ denotes the operator associated with C, i.e., $\star$ if $C \in \mathfrak{C}_\star$, else $\oplus$ if $C \in \mathfrak{C}_\oplus$, and otherwise $\vee$.[3] In case $[\![C]\!]_\circ|_\delta = \mathbf{0}$, we have $[\![\mathfrak{C}]\!]|_\delta = \mathbf{0}$, which is already covered in the sum. From this it directly follows that at most two additional functions have to be represented for each clause in the i-th cutset. Thus, at most

$$3 + \sum_{i=0}^{k-2} 2^{w(\mathsf{Cutset}_{\mathcal{H}}(X_i^\pi))}$$

distinct functions are represented in a BDD, providing an upper bound on the number of nodes in the BDD for $\mathfrak{C}$.

We now prove the remaining claims (1) and (2). Let δ be any partial configuration with $\mathsf{Dom}(\delta) = X_i^\pi$. The clause operators $\vee$ and $\oplus$ enjoy the property that under δ, the subset $X_i^\pi \cap C$ of clause literals can be replaced by the Boolean constant $[\![X_i^\pi \cap C]\!]_\circ(\delta) \in \{0, 1\}$ while retaining the semantics. The same replacement can also be done for the operator $\star$, unless $\sum_{\ell \in X_i^\pi \cap C} \delta(\ell) > 1$ and thus $[\![C]\!]_\circ|_\delta = \mathbf{0}$. Then for case (1), $[\![C]\!]_\circ|_\delta$ can only be one of two possibly non-$\mathbf{0}$ Boolean functions, the one replacing $X_i^\pi \cap C$ by 0 and the one replacing $X_i^\pi \cap C$ by 1. Considering case (2), C either does not contain any instantiated variables, i.e., $X_i^\pi \cap C = \varnothing$, then $[\![C]\!]_\circ|_\delta = [\![C]\!]_\circ$. Or all variables in C are instantiated, i.e., $C \subseteq X_i^\pi$, in which case $[\![C]\!]_\circ|_\delta \in \{\mathbf{0}, \mathbf{1}\}$. This shows the claims (1) and (2) and thus the upper bound of the theorem. □

Clause Ordering. Next to variable ordering, the order of clauses towards a left-deep BDD compilation (see Algorithm 1) has shown great impact on the intermediate BDD sizes and hence construction performance [4, 48]. Intuitively, reducing the total span between clauses that share common variables fosters the exploitation of BDD reductions and yields fewer changes of variable dependencies during BDD construction. An extremal illustrative example is the CNF formula $z \wedge \varphi_m \wedge \neg z$ with the linear clause order ρ. While the BDD representing this CNF is $\boxed{0}$, a left-deep BDD compilation according to Algorithm 1 constructs the potentially exponential BDD for φ_m. Minimizing the total span and thus bringing clauses z and $\neg z$ close together enables to reach to $\boxed{0}$ earlier, preferably at the beginning of the construction.

3.2.2. Hypergraph-based Heuristics

Most successful static ordering heuristics for CNFs depend on analyzing their variable or clause hypergraph. Recall that in such hypergraphs $\mathcal{H} = \langle V, E, w \rangle$, vertices V represent either variables or clauses, and hyperedges E model the dependencies between variables or clauses, respectively. FORCE [3] and MINCE [4] are such heuristics reducing the *total span*.

[3] A clause $C \in \mathfrak{C}$ may in principle be contained in multiple of $\mathfrak{C}_\vee$, $\mathfrak{C}_\oplus$, and $\mathfrak{C}_\star$. Still, the associated operator of C is a single one: recall that a ONE-HOT clause C subsumes the XOR clause C, which in turn subsumes the disjunctive clause C. Any subsumed clauses can be removed from $\mathfrak{C}$ without changing the semantics $[\![\mathfrak{C}]\!]$.

Algorithm 4: ReMINCE($\mathcal{H}, \varepsilon$)

input : hypergraph $\mathcal{H} = \langle V, E \rangle$, imbalance $\varepsilon \in (0, 1/2)$
output: total order $o : \mathbb{N}_{<|V|} \to V$

Procedure BiRec(L, M)
  input : $L, M \subseteq V$
  output: total order $\tau : \mathbb{N}_{<|M|} \to M$
  if $M = \{x\}$ **then return** $\{0 \mapsto x\}$
  $B \coloneqq \varepsilon$-BalancedMincut$_{\mathcal{H}}(L, M)$
  return BiRec(L, B) ++ BiRec($L \cup B, M \setminus B$)
return BiRec($\varnothing, V$)

FORCE. Initialized with a random order, FORCE iteratively updates a current order $\pi : \mathbb{N}_{<|V|} \to V$ towards a new order π'. For this, the *center of gravity* $\mathsf{cog}(e) \coloneqq \sum_{v \in e} \pi^{-1}(v)/|e|$ is computed for each edge $e \in E$. Then, the tentative position $p(v)$ for a vertex v in the updated order π' is given by the weighted average

$$p(v) \;\coloneqq\; \frac{\sum_{e \in E_v} w(e) \cdot \mathsf{cog}(e)}{\sum_{e \in E_v} w(e)}$$

where $E_v \coloneqq \{e \in E \mid v \in e\}$ denotes the set of hyperedges containing v. The new order π' is chosen such that it agrees with p, i.e., $p(x) \leqslant p(y)$ for any two vertices $x, y \in V$ with $\pi'(x) \leqslant \pi'(y)$. Additionally, if $p(x) = p(y)$, then we require that π' agrees with π, formally $\pi(x) \leqslant \pi(y)$. This is implemented via a simple sorting procedure. Setting the updated order to the current one, FORCE iterates until it either stabilizes, a quality metric such as the total span stops decreasing, a high number of iterations has been performed, or a predefined time limit has been reached [3]. We opt for iteration until stabilization and select the order with the lowest total span found along the way.

MINCE. The main idea for MINCE is to recursively determine ε-balanced mincuts until reaching singleton cuts. Due to the (nearly) minimal cutting, dependent nodes are likely to not be separated by the first cuts and are thus placed close to each other, reducing the overall total span. While FORCE can easily be implemented following the pseudocode of its original publication [3], MINCE has not been fully specified. Notably, MINCE is only available as binary executable that includes the circuit placer CAPO used to compute approximate balanced mincuts with predefined parameters [15]. To investigate the impact of state-of-the-art cutting algorithms and parameters on the performance of MINCE, we implemented *ReMINCE* following the pseudocode provided in Algorithm 4. Our reimplementation can take advantage of state-of-the-art hypergraph cutting tools such as KaHyPar [53] and vary parameters such as the imbalance factor ε and hypergraph weights.

Note that hypergraph cutters typically do not directly incorporate the concept of a left-context as used in our definition of balanced mincuts. However, they allow specifying vertex weights and fixing vertices to partitions. Contracting the left-context L and the right-context $R = V \setminus (M \cup L)$

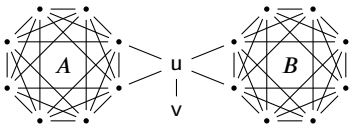
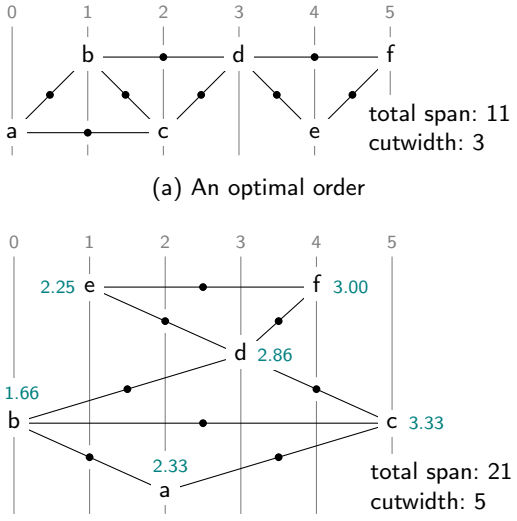


Figure 4: Example of a hypergraph where mincuts with unbounded imbalance yield a suboptimal order.



(a) An optimal order

(b) Suboptimal order, which FORCE fails to improve. Tentative positions computed by FORCE are next to each vertex in teal.

Figure 5: Example hypergraph where MINCE yields an optimal order, but FORCE may get stuck with a suboptimal order. Vertices are labeled by letters *a* to *f*. Vertex positions are numbered from left to right. Dots on hyperedges indicate their center of gravity. Vertical placement of nodes has no meaning.

each into one pseudo-vertex with weight 0 and fixing them to a left- and right-partition, respectively, emulates our setting towards balanced mincuts.

Imbalance Factors in MINCE. Recall that the mincut imbalance factor ε gives an upper bound on the size of the larger partition. As an example, an ε close to $1/2$ restricts the larger partition to contain at most $3/4$ of the nodes, while an ε close to 0 requires the two partitions to be equally large (modulo rounding). When allowing a larger imbalance, the costs of a single cut can potentially be reduced. However, this does not necessarily translate to a better variable or clause order. To keep the upper bound from Theorem 1 on the BDD size low, the primary goal is to minimize the *maximal* cutset size. There is no value in making a cheaper cut at the beginning if that implies a more expensive cut later on. After all, every vertex is placed in its own partition at the end of the recursive bipartitioning procedure. As an example, consider the hypergraph illustrated in Figure 4, consisting of two vertex clusters A and B that itself are densely connected. Additionally, there are two vertices u and v, where u is connected to v as well as two vertices in A and two vertices in B via separate hyperedges each. Apart from that, v is not connected to any other vertex, and there is also no hyperedge connecting vertices of A and B. Clearly, u and v should appear somewhere in the middle of the order. Without a bound on the imbalance, however, the first mincut will split v apart, causing it to be placed at one of the order's ends and making subsequent mincuts more expensive.

Comparing FORCE and MINCE. FORCE is the de facto standard for ordering variables due to its simple implementation and good performance [3, 28, 48]. In the original implementation of FORCE and MINCE, the former was faster in determining an order than the latter. However, experiments showed that on larger compilation instances, MINCE usually leads to smaller BDD sizes and faster BDD construction than FORCE [3]. That FORCE admits a more fundamental drawback can be illustrated already with small examples like in Figure 5. Here, the hypergraph consists of two vertex clusters a to d and d to f, both sharing the node d. Depending on the initial order, the FORCE algorithm may eventually encounter a vertex order where the two clusters are interleaved as depicted in Figure 5b. In this order, the vertices are sorted by their tentative positions, meaning that FORCE terminates here. However, the returned order is only a local optimum regarding the total span and the cutwidth. In contrast to FORCE, MINCE minimizes cuts in the hypergraph and hence directly separates the two clusters. Thereby, it avoids the costly interleaving and yields a globally optimal order with almost half the total span and cutwidth. Note that such constellations generalize well to larger hypergraphs: averaging towards tentative positions in FORCE does not directly take cutsets into account and can thus place densely-connected vertices far from each other.

3.3. Compilation Process

The previous sections concerned preprocessing of the input for the BDD construction following Algorithm 1. In this section, we discuss techniques for the compilation process itself, compiling an input XCNF into a BDD using different BDD engines and construction schemes.

3.3.1. Clause Bracketing

As conjunction is an associative operation, there are different ways to bracket clauses in an XCNF without changing its semantics. The exact bracketing is reflected in the abstract syntax tree of the XCNF, whose structure is crucial for the BDD construction [48, 61]. We consider two different bracketing schemes: *left-deep* and *balanced*. A left-deep bracketing groups left-most clauses, e.g.,

$$\big((C_0 \wedge C_1) \wedge C_2\big) \wedge C_3,$$

and is used in the standard construction as given in Algorithm 1 [4, 23, 28, 16, 22]. During such a construction, the BDD of the left operand is likely to grow fast during the construction, while the right operand covers only one clause and is thus comparably small. Therefore, many conjunctive operations are expected to operate on large BDDs [16]. Hence, methods to balance the size of conjoined clauses and hence BDD sizes were developed [48, 61, 16], interpreting

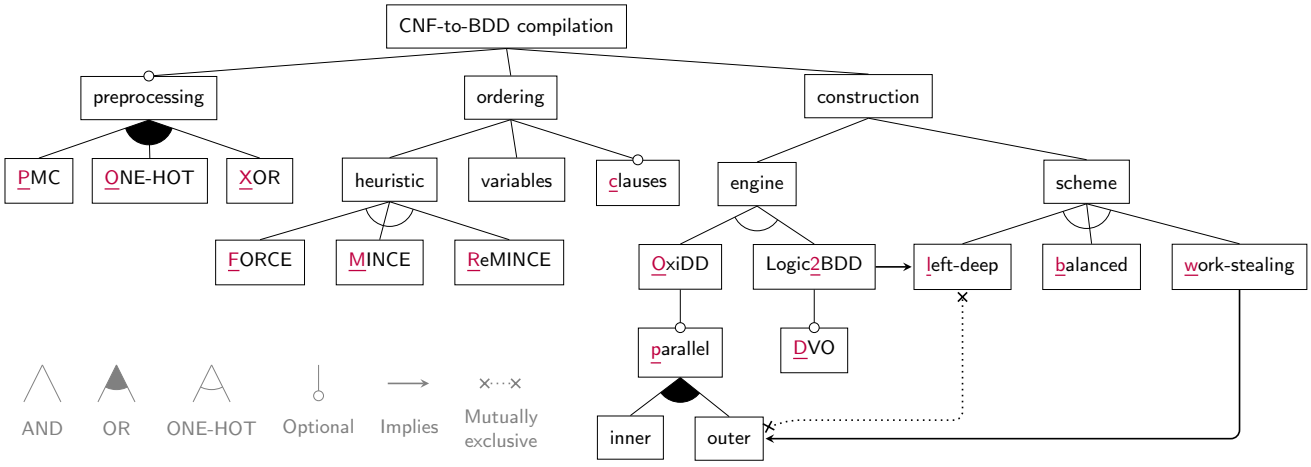


Figure 6: Feature diagram for CNF-to-BDD compilation configurations.

the ordered clauses as an approximately balanced tree. Following the above example, a balanced bracketing would be

$$(C_0 \wedge C_1) \wedge (C_2 \wedge C_3).$$

Such a bracketing can be achieved by performing recursive bracketing, each splitting the XCNF into two parts of similar size.

3.3.2. Parallelization

To speed up BDD construction on multicore processors, one option is to parallelize the conjunction operations during incremental bottom-up construction using *work-stealing* [10]. Here, the recursive calls for the *then* and *else* branches in `apply_and` are executed concurrently. In the context of BDDs, this idea has been pioneered by SYLVAN [20], with OXIDD [31] supporting this form of operation-internal parallelization as well.

Besides this form of *inner parallelism*, also multiple conjunction operations in independent parts of the construction tree could be executed concurrently. We call this form of concurrent execution *outer parallelism*. Instead of fixing a clause bracketing in advance, we can also allow free reassociation of conjunction operations at runtime, while maintaining the clause order (see section above). Intuitively, the latter approach assumes intermediate BDD construction times to increase when their sizes do. Eagerly conjoining intermediate BDDs right after their computation might hence also reduce the aforementioned size drift during left-deep construction. Outer and inner parallelization approaches can be freely combined.

3.3.3. BDD Engine

The performance of CNF-to-BDD compilation also depends on the chosen BDD library that maintains the data structure and performs the basic `apply` operations [25]. State-of-the-art libraries are, e.g., BUDDY [42], CUDD [56], SYLVAN [20], and OXIDD [31]. The latter two also support parallelization of `apply` operations. Dedicated CNF-to-BDD compilers implementing incremental construction usually rely on CUDD. In the context of compiling feature models, LOGIC2BDD [23] uses CUDD as backend and modifies its *dynamic variable reordering* (DVO) [52] for a dedicated treatment of feature variable groups. With this enhancement, LOGIC2BDD enabled the compilation of large feature models [28, 58]. OXIDD has shown to be en par or even outperform other BDD libraries and ships with a command-line interface OXIDD-CLI. The latter supports CNF-to-BDD compilation where different construction schemes and OXIDD options can be configured [31]. We consider LOGIC2BDD and OXIDD without any advanced construction techniques as the basic settings, i.e., without DVO and parallel construction, respectively. The respective advanced construction mechanisms can then be enabled on demand.

4. Feature Model Compilation

In this section, we empirically evaluate the impact of CNF-to-BDD compilation techniques when applied on CNFs for feature models. Here, we focus on the techniques summarized in the last section. Our experiments and evaluation are driven by the following research questions:

(RQ1) How is CNF-to-BDD compilation performance for feature models affected by different
- **a)** preprocessing techniques,
- **b)** variable or clause ordering,
- **c)** processing engines and schemes, and
- **d)** parallelization?

(RQ2) How does optimizing construction techniques impact BDD compilation on large-scale feature models?

The first research question mainly focuses on the exploration of BDD compilation techniques from the literature. The second then asks for the utilization of insights about such techniques to potentially obtain BDDs for feature models that the literature considered too complex to compile.

4.1. Configuring CNF-to-BDD Compilation

Given the manifold different techniques and tunable parameters for CNF-to-BDD compilation, we model the compilation itself as a configurable system. This leads to a feature model for CNF-to-BDD compilation as given by the feature diagram in Figure 6. Here, we take on a simplified view on configurations, modelling only those that are relevant for our experiments. In particular, we neglect the possibilities to not apply any variable ordering, or to use different heuristics for variable and clause ordering. Deviating from our experiment setup, the feature model permits enabling OxiDD's inner and outer parallelism independently. This distinction is solely for explanatory purposes. For the evaluation, we either enable none or all possible parallelism options. Note that the feature diagram contains three cross-tree constraints: Logic2BDD (`2`) only supports a left-deep construction scheme (`1`) while OxiDD's outer parallelism is incompatible with this scheme, leading to `1` and outer parallelism mutually excluding each other. Work-stealing (`w`) on the other hand requires outer parallelism. Overall, this leads to $2^3 \cdot (3 \cdot 2) \cdot (2 \cdot (1{+}2{+}1) + 2) = 480$ different possibilities to configure CNF-to-BDD compilation.

We denote configurations as triples of *preprocessing*, *ordering*, and *construction* options. For instance, "`px-mc-opb`" refers to the following configuration: The initial CNF is preprocessed using PMC (`p`) and XOR factorization (`x`). Then, MINCE (`m`) is used as the ordering heuristic to generate a variable as well as a clause order (`c`). Finally, OxiDD (`o`) performs a parallelized (`p`) construction of the feature model using a balanced (`b`) construction scheme. When no preprocessing is applied, we denote this as ∅, e.g., "`∅-f-21`" refers to a left-deep (`1`) construction using Logic2BDD (`2`) with FORCE (`f`) variable ordering, but without clause ordering and without any preprocessing. Note that `p` and `o` are used in both the *preprocessing* and *construction* components. Their meaning will be clear from the context or their placement in the configuration triple.

Implementation. We implemented the techniques for the CNF-to-BDD compilation configurations in a tool called Dimagic 2.0 [33] written in Rust. The different CNF-to-BDD compilation configurations can be enabled individually via command line flags. Our tool takes DIMACS files as input, the standard file format for CNF formulas obtained from feature models using FeatureIDE [60]. The output is an extended DIMACS file format that supports XCNF with ONE-HOT and XOR groups. Variable and clause orders are encoded as supported by FeatureIDE [60], i.e., variable names in comments and the order determined by the sequence of lines listing variables and clauses. PMC and MINCE are called as external binaries [41, 2]. ONE-HOT and XOR factorization, as well as FORCE and ReMINCE are implemented in the tool itself, the latter employing the state-of-the-art hypergraph cutting library KaHyPar [53]. The hypergraph cutters in MINCE and ReMINCE use probabilistic algorithms and incorporate a seedable random number generator, while FORCE is deterministic given an initial order of hypergraph nodes.

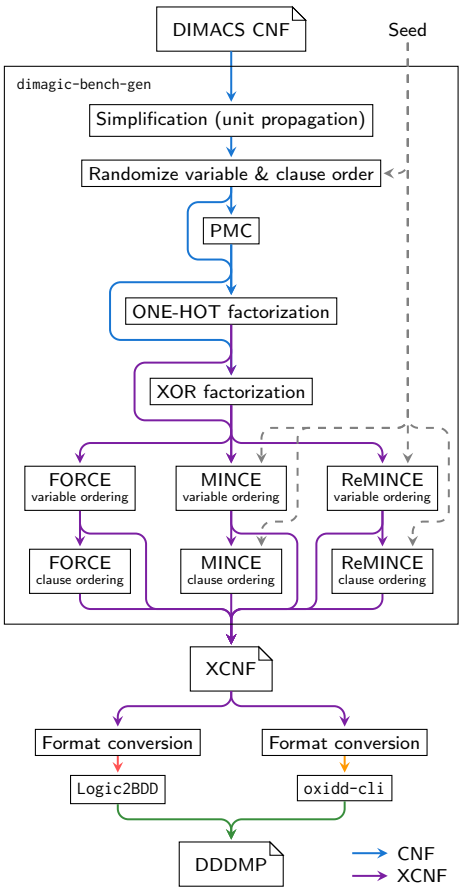


Figure 7: Experiment pipeline.

4.2. Experiment Setup

Experiment Pipeline. While `dimagic` is intended as reusable preprocessing tool, `dimagic-bench-gen` provides a frontend specifically targeted at generating XCNFs for a set of configurations. Its pipeline is shown in Figure 7. Note that we always perform an initial unit propagation pass to enable a fairer comparison among the other more costly preprocessing techniques. Unit propagation also has a very good cost-utility ratio, being very efficient and simplifying the CNF structure. Reductions by more than an order of magnitude in the number of clauses are well possible (see Table 3). Our pipeline involves probabilistic algorithms, e.g., ReMINCE or FORCE. To achieve a certain degree of robustness in our evaluation, we run the experiment pipeline for five fixed random seeds each and typically report the median for compilation times. In this vein, we call a model *median-constructible* if the compilation succeeded for at least three out of five seeds.

For BDD compilation, we use OxiDD-CLI 0.3.0[4] and Logic2BDD.[5] The underlying libraries CUDD and OxiDD are configured to use complement edges in BDDs. We patched Logic2BDD to support ONE-HOT groups with negative literals as well. Still, Logic2BDD does not support

[4] `https://github.com/oxidd/oxidd`

[5] `https://github.com/davidfa71/Extending-Logic`, as of commit `63ce01c`

dedicated construction of XOR groups. We export the resulting BDDs in DDDMP format, the standard file format to store BDDs originally introduced in CUDD [56]. To test correctness of our compilations, we compute model counts with OxiDD-CLI using arbitrary-precision integers, which we cross-check with model counts obtained from the #SAT solver SharpSAT-TD [40].[6]

Experiment System. All experiments were conducted on an AMD Ryzen 9 5950X with 16 physical cores and 128 GiB RAM running Ubuntu 24.04 (Linux kernel 6.8). We set a timeout of 10 minutes and impose a memory limit of roughly 4 GiB in OxiDD by restricting the node count to 140,928,616 and the *apply cache* to 33,554,432 entries. Logic2BDD does not expose fitting memory settings. Hence, we do not evaluate memory consumptions. To ensure that multiple experiments on different cores do not interfere with respect to memory constraints, we enforce a memory limit of 7 GiB for each instance via Linux control groups. We perform the #SAT computations on BDDs generated from DDDMP files in a separate step because of memory requirements, given that the arbitrary precision integers stored for every BDD node can easily demand a multiple of the BDD size.

4.3. Impact Analysis (RQ1)

In our configuration space of CNF-to-BDD configurations, we already reduced the amount of CNF-to-BDD configurations, e.g., by disallowing mixtures of different ordering heuristics (see Section 4.1). However, there are still too many configurations to analyze them all at once. We hence perform a stepwise impact analysis of each of the four categories of **RQ1**, also taking insights from previous simplified experiments into account [21].

For answering **RQ1**, we conduct experiments using the benchmarks of the UnWise feature model sampler [26] as target models. This benchmark set comprises 49 representative state-of-the-art feature models that cover a broad spectrum of different models and sizes. Specifically, we conduct the following experiments:

(E1) Towards answering **RQ1a** we first determine the impact of preprocessing onto BDD construction time and size. Here, we stepwise add preprocessing techniques to the configuration `rc-ob`[7]—see Section 4.3.1.

(E2) On the best preprocessing configuration obtained from Experiment **E1**, we investigate the impact of mincut parameters ϵ and hypergraph weights onto ReMINCE, comparing them with the classic MINCE implementation, contributing to **RQ1b**—see Section 4.3.2.

(E3) Relying on the best configurations obtained from Experiments **E1** and **E2**, we compare the performance of different engines, construction schemes, and FORCE and ReMINCE orderings. This experiment targets **RQ1a-c**—see Section 4.3.3.

(E4) Based on the best-performing configuration of the previous experiments, we vary construction schemes and analyze the impact of multi-threaded construction and `apply` operations in OxiDD, targeting **RQ1c-d**—see Section 4.3.4.

4.3.1. Preprocessing (E1)

To assess the impact of preprocessing onto compilation performance, we investigate the configurations `*-rc-ob`, i.e., sequential balanced compilation using OxiDD with ReMINCE variable and clause ordering. We chose 16 as an upper bound of the XOR recovery clause size to achieve a reasonable trade-off between recovery success rate and time. Additionally, we set a timeout of 5 min, which was only reached for `embtoolkit-smarch` without PMC. Notably, while XOR factorization has shown great benefits for SAT solving [57], this barely is the case for BDD compilation of feature models. With PMC enabled, all recovered XORs are subsumed by ONE-HOTs in our experiments. In particular, the configurations `pox` and `po` are effectively the same. Hence, we ignore `pox` in the sequel.

Table 1 shows how many of the 49 UnWise models could be compiled for the different preprocessing configurations with at most two failures for five random seeds. It appears that PMC in combination with ONE-HOT factorization (`po`) performs best. Figure 8 details how the median compilation times change when disabling either PMC or ONE-HOT factorization, or replacing ONE-HOT by XOR factorization. The general impression is that larger models (with compilation times of at least one second) benefit from PMC and ONE-HOT factorization. Especially the improvements of PMC are remarkable, with nine additional constructible models and six larger models that could be compiled much faster with speedups ranging from 2.5 to 146. Only two models take considerably longer to compile with PMC preprocessing: `financialServices01` and `fiasco`. The improvements of ONE-HOT factorization are more moderate, still enabling the compilation of two additional models.

PMC and ONE-HOT factorization are also very beneficial with respect to the time needed to compute variable and clause order. This is due to vast reductions in the number of clauses (cf. Table 3), and simplifications in the XCNF structure. Without any preprocessing, ReMINCE takes up to 296 s to compute a variable order for the `automotive2_4` model. This is the longest computation time among models we have been able to compile. Clause orders tend to be even more costly to compute, the longest observed computation time is 2,516 s, again for `automotive2_4`. In contrast, PMC preprocessing only takes 46 s for `automotive2_4`, and less than 4 s for every other constructible model. ONE-HOT factorization is generally very fast with less than 0.3 s regardless of

[6] Within 10 minutes, computing the SAT count succeeded for all models but `linux-2.6.33.3`, for which we could not compile a BDD either [59].

[7] We choose OxiDD in favor of Logic2BDD as baseline engine, since the latter supports ONE-HOT but not XOR group construction.

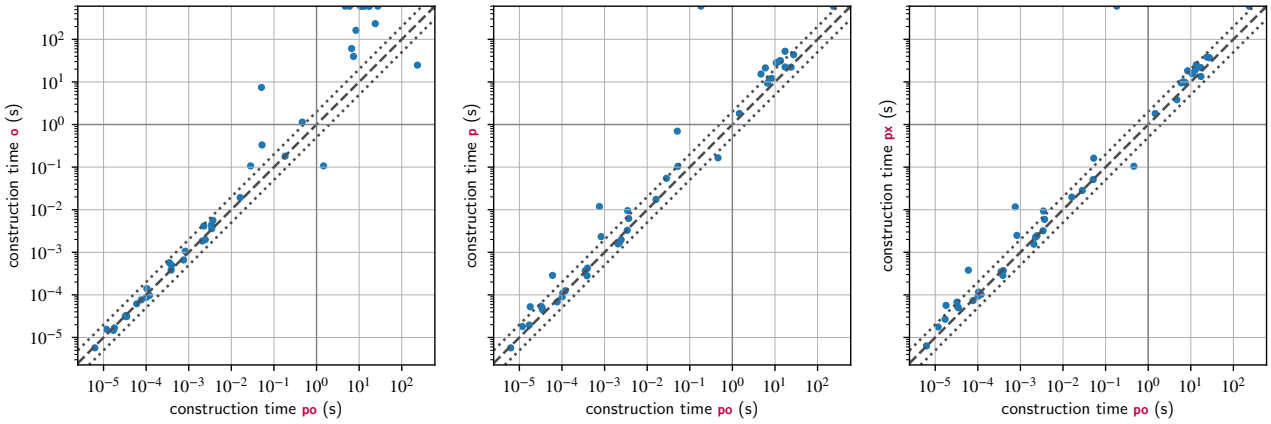


Figure 8: Impact analysis of preprocessing techniques on UnWise models. Timings are the median over five runs with different random seeds. Dotted lines indicate a factor 2 or 1/2, respectively, compared to the baseline.

Table 1
Median-constructible UnWise models by preprocessing.

Preprocessing	∅	x	o	ox	p	px	po	pox
Constructible Models	34	33	35	34	42	42	44	44

whether PMC is enabled or not.[8] With both PMC and ONE-HOT factorization enabled, the longest observed variable ordering time is 51 s for `automotive2_4` (again, only taking constructible models into account). The longest ReMINCE clause ordering time is 38 s for constructible models, with `financialServices01` taking longest. The overall preprocessing time (including variable and clause ordering) is lower in the `po-rc` configuration compared to `∅-rc` for all UnWise models, with a speedup ranging from 1.1 to 48.3 (geometric mean: 2.7). The longest overall preprocessing time is 108 s for constructible models (`automotive2_4`). All eCos models could be preprocessed within 21 s each. Given how many models could be compiled with `po-rc` preprocessing only, these costs pay off easily.

Answering **RQ1a**, we find that CNF-to-BDD compilation performance greatly benefits from PMC preprocessing and ONE-HOT factorization, while XOR factorization is negligible for feature models.

4.3.2. MINCE vs. ReMINCE (E2)

The main drawback of the original MINCE implementation [4] is occasional failure to generate orders [21], but also its mere availability as a binary with consequently less configurability and unclear algorithmic aspects. This motivates a reimplementation of MINCE. Following Algorithm 4, we integrated our reimplementation ReMINCE in Dimagic 2.0. In this experiment, we compare the performance of MINCE with several configurations of ReMINCE based on a balanced OxiDD construction with best preprocessing. That is, we focus on configurations `po-mc-ob` and `po-rc-ob`.

We consider three versions of the variable hypergraph $\mathcal{H}(\mathfrak{C}, b_\vee, b_\oplus, b_\star)$ with different base weights each:

(a) $b_\vee = 1,\ b_\oplus = 1,\ b_\star = 1$

(b) $b_\vee = 1,\ b_\oplus = 2,\ b_\star = 2$

(c) $b_\vee = 1,\ b_\oplus = |\mathfrak{C}|,\ b_\star = |\mathfrak{C}|$

Note that we use equal base weights for ONE-HOT and XOR clauses, since `po` preprocessing only produces XOR clauses of size 2, i.e., they can also be viewed as ONE-HOT clauses. Option (c) strictly prioritizes ONE-HOT clauses, ensuring that the span for such clauses is as low as possible, potentially at the cost of higher spans for disjunctive clauses. This is hence in the spirit of the recently presented FORCE_{xg} heuristic [27]. Option (a) with unit base weights is motivated by the upper bound on the BDD node count from Theorem 1. The slightly biased option (b) is based on the motivation for option (a), but incorporates the observation that an XOR or ONE-HOT clause over k variables has $k-1$ additional BDD nodes compared to a disjunctive clause.

As a further configuration option of ReMINCE, we consider three imbalance factors $\varepsilon \in \{0.01, 0.1, 0.4999\}$. Unlike the clause weights, which only apply to the variable hypergraph, the imbalance factor is relevant for both cutting the variable and the clause hypergraph. In principle, there is no need to choose the same ε for computing variable and clause orders, but we do not consider that additional dimension in our evaluation.

Experiment Results. First of all, we remark that PMC and ONE-HOT factorization successfully mitigate the runtime errors of the original MINCE, apparent without any preprocessing [21]. Thus, a fair comparison between MINCE and ReMINCE is possible on preprocessed models. An initial

[8] This even includes the five models we have not been able to compile. XOR factorization is even faster with less than 0.06 s provided that PMC is enabled. With PMC disabled, XOR factorization takes up to 1.1 s for constructible UnWise models, but may take more than 300 s for the remaining ones.

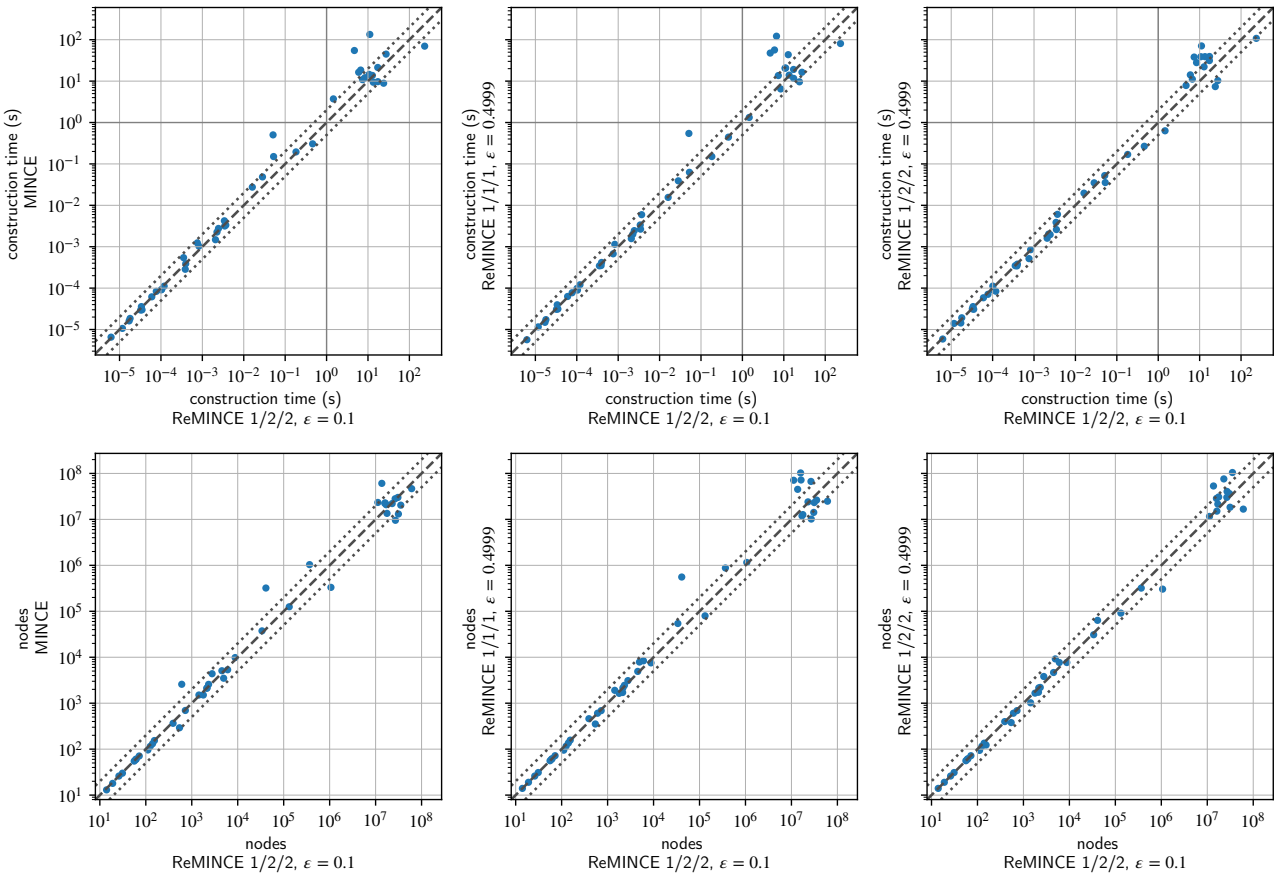


Figure 9: MINCE and other ReMINCE configurations compared to the best ReMINCE configuration (on the x-axis) for UnWise models. The first row shows construction times as medians over five runs with different random seeds, the second row the medians of the resulting BDDs' node counts. Base weights are denoted as triples $b_\vee/b_\oplus/b_\star$. Data points for the five non-constructible models are excluded.

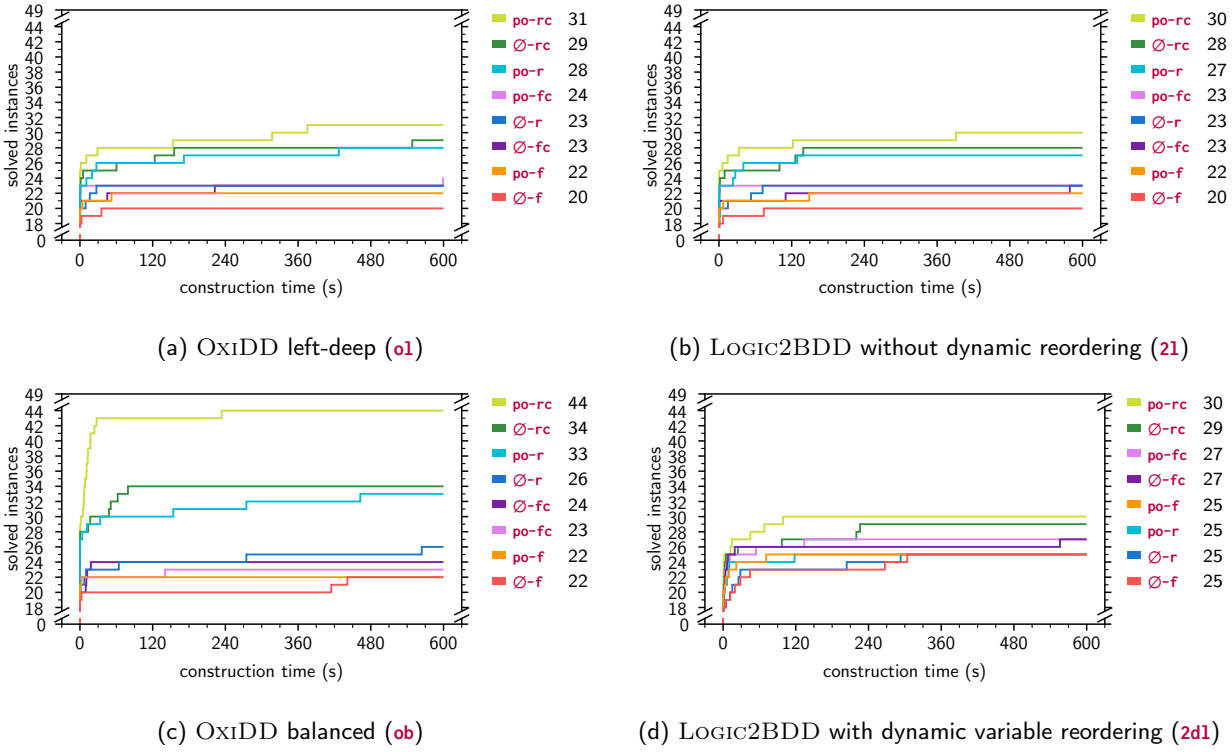
Table 2
Performance of MINCE and various ReMINCE configurations on the UnWise suite. Medians are taken across runs with five different seeds. For the compilation times, each model with three or more compilation failures counts as 600 s.

		Compilation Failures								
		Median			Total			∑ Median Times (s)		
Heuristic	ε	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)
MINCE		5			34			3,444		
ReMINCE	0.01	16	17	18	77	74	88	9,741	10,301	11,347
ReMINCE	0.1	6	5	6	38	29	38	3,993	3,403	4,521
ReMINCE	0.4999	5	5	9	33	36	53	3,485	3,468	6,404

observation is that ReMINCE performs worst with the smallest imbalance $\varepsilon = 0.01$, with more than 30 % compilation failures (cf. Table 2). Further, strictly prioritizing ONE-HOT and XOR clauses as done with the $\hat{w}$ weights tends to be worse than giving them roughly equal weight as disjunctive clauses. This aligns with the theoretical upper bound on BDD sizes from Theorem 1, which is agnostic of the clause weights. So while ONE-HOT clauses deserve special care in XCNF preprocessing (cf. Section 4.3.1), their role is not as special with respect to variable ordering.

The number of compilation failures shown in Table 2 suggests that ReMINCE with $\varepsilon = 0.1$ and clause weights $\bar{w}$, i.e., weight 2 for ONE-HOT and XOR clauses, gives the best results. Out of the 245 compilations in total (49 models times 5 runs with different seeds), only 29 timed out or ran out of memory. Given that five models could not be compiled in any configuration and for any seed, this means that the compilation is quite robust, with four failures distributed across the remaining $44 \cdot 5$ compilations. When taking the median across the five seeds, i.e., considering a model as failed iff three or more out of five compilation failed, we observe that only the five most challenging models cannot be compiled. The latter also holds true for three other configurations: (1) MINCE, (2) ReMINCE with $\varepsilon = 0.4999$ and uni-weights (w) as well as (3) ReMINCE with $\varepsilon = 0.4999$ and weight 2 for ONE-HOTs and XORs ($\bar{w}$).

A detailed comparison between these configurations and the best one as a baseline is shown in Figure 9. Most models



(a) OxiDD left-deep (`ol`)

(b) Logic2BDD without dynamic reordering (`2l`)

(c) OxiDD balanced (`ob`)

(d) Logic2BDD with dynamic variable reordering (`2dl`)

Figure 10: Impact analysis of single-threaded configurations. Timings are the median over five runs with different random seeds. The number of successfully constructed instances after 600 s are on the right of each plot.

have roughly the same construction time for all four configurations, but there is a considerable amount of outliers for larger models, where the baseline performs better. The best configuration mainly struggles with the `financialServices01` model (the point at the top right of each plot in the first row). As a further quality metric for the variable order, we consider the final BDD node count (cf. the second row of Figure 9), but remark that the results are inconclusive.

Table 2 also presents cumulative median compilation times, where each failed model is penalized with 600 s, i.e., the timeout duration. By taking the sum across models, we intentionally give more weight to the larger models. Here again, ReMINCE with $\varepsilon = 0.1$ and weight 2 performs best.

Regarding **RQ1b**, we find that ReMINCE is competitive with MINCE. The best ReMINCE configuration uses slightly biased clause weights and a moderate imbalance factor. It is more robust than MINCE in terms of compilation failures.

4.3.3. Engines, Orderings, and Construction (E3)

To more broadly evaluate the impact of configuration options on BDD compilation performance, we used FORCE variable ordering as a baseline (`f`) and compare with ReMINCE variable ordering (`r`). Both were combined with clause ordering using the very same ordering heuristic, i.e., `fc` and `rc`, and the preceding preprocessing configuration `po` following the best-performing configuration determined in Section 4.3.1. This leads to eight base configurations: `∅-f`, `po-f`, `∅-fc`, `po-fc`, `∅-r`, `po-r`, `∅-rc`, and `po-rc`. For each of these base configurations we consider four different engine options: `ol`, `ob`, `2l`, and `2dl`.

Figure 10 shows for each of the four engine configurations a plot on how many of the 49 models could be constructed up to a given time of 10 minutes with the eight different base configurations. For the timings, we chose the median over five compilations with different random seeds. Thus, Figure 10 contains data points for configurations with at most two compilation failures. Preprocessing times are not included, since we would need to tune the options of the preprocessing tools—especially the hypergraph cutter used in ReMINCE—to thoroughly assess the trade-off between preprocessing and construction times. However, we remark that the preprocessing times are reasonably low for the best-performing configuration `po-rc` (cf. Section 4.3.1).

Figures 10a and 10b correspond to the basic settings of the BDD engines, where both OxiDD and Logic2BDD are used to perform a left-deep single-threaded construction. OxiDD has slight advantages here due to its superior performance compared to CUDD, the backend BDD library of Logic2BDD [31]. Figure 10c and Figure 10d show optimizations unique to OxiDD and Logic2BDD, i.e., balanced construction and component-wise dynamic variable reordering [52], respectively. The balanced construction scheme greatly improves performance for the better-performing configurations, with up to 13 additional models

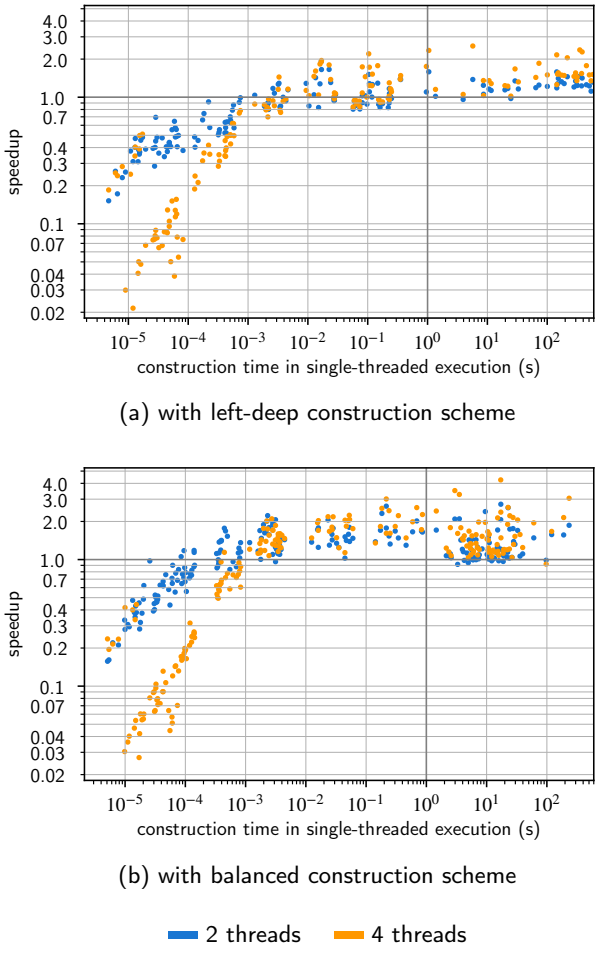
Table 3
OxiDD balanced (ob) construction times in seconds for the models of the UnWise benchmark suite (t: timeout [> 600 s], m: memout [> 4 GiB], t/m: at least once each timeout and memout). All eCos models contained in the UnWise suite are marked with "*". For many models, the clause count after PMC (p) varies depending on the seed. Here, $a{+}b$ denotes the interval $[a, a+b]$.

Name	Input		After UP	po		∅-f		po-f		∅-fc		po-fc		∅-r		po-r		∅-rc		po-rc	
	#vars	#clauses	#clauses	#clauses	#OH	max	min	max	min	max	min	max	min	max	min	max	min	max	min	max	min
x264	16	11	11	5	2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
dune	17	16	16	8	2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
berkeleydbc	18	29	20	4	2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
lrzip	20	63	49	8	6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
hipacc	31	104	104	68	2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
javagc	39	105	67	13	6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
polly	40	100	63	18	9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
vp9	42	104	63	17	8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
7z	44	210	167	9	4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
jhipster	45	104	82	54	8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
berkeleydb	76	141	140	92+1	13	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
axtls_2_1_4	94	190	190	96+8	27+1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
axtls-kconfig	96	203	89	66	15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
fiasco_17_10	234	1,178	1,141	626+27	37+5	t	37.90	3.94	1.05	t	0.26	1.55	0.05	t	69.45	18.03	6.75	0.93	0.14	0.34	0.03
uclibc-ng_1_0_29	269	1,403	1,403	578+7	33+1	5.41	0.33	1.13	0.08	0.08	0.03	0.03	0.01	1.29	1.00	1.20	0.30	0.04	0.01	0.02	0.01
uclibc	313	1,285	933	221	24	0.03	0.02	0.00	0.00	0.01	0.00	0.00	0.00	0.04	0.02	0.00	0.00	0.01	0.00	0.00	0.00
toybox_0_7_5	316	108	93	87	4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
uclinux-base	380	7,366	3,562	148	33	420.43	284.82	0.00	0.00	14.01	5.03	0.00	0.00	t/m	386.38	0.00	0.00	0.02	0.01	0.00	0.00
toybox	544	1,020	840	519+11	63+5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
axtls-smarch	684	2,155	2,041	744+32	90+10	7.37	0.12	0.01	0.00	0.04	0.02	0.00	0.00	13.08	3.19	0.00	0.00	0.01	0.00	0.00	0.00
financialServices01	771	7,241	7,141	4,254+50	63+7	t/m	t/m	t/m	t/m	m	m	m	m	t/m	t/m	t/m	t/m	18.51	2.86	t/m	3.53
busybox-1.18.0-kconfig	854	1,322	659	560	31	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
busybox_1_28_0	998	962	962	784	28	t/m	5.50	t/m	19.04	m	0.87	t/m	4.45	0.01	0.01	0.01	0.00	0.01	0.00	0.00	0.00
am31_sim*	1,178	2,845	2,258	1,419+51	518+17	t	t	t	t	t	t	t	11.79	t/m	68.31	t/m	14.25	m	22.44	120.07	3.73
embtoolkit-kconfig	1,179	5,967	4,435	1,182+16	97+6	t/m	t/m	t/m	34.80	m	2.12	t/m	11.58	9.25	6.77	0.01	0.01	0.02	0.01	0.00	0.00
ref4955*	1,218	3,099	2,385	1,518+7	534+3	t	t	t	t	t/m	t/m	t	t	t/m	t/m	t	115.38	t/m	t/m	22.32	4.10
ecos-icse11	1,244	3,146	2,445	1,546+9	565+3	t	t	t	t	t	t	t/m	t/m	t/m	t/m	t	44.24	m	38.98	15.89	2.50
pati*	1,248	3,266	2,491	1,542+11	555+4	t	t	t	t	t/m	t/m	t/m	t/m	t/m	87.42	t	39.05	t/m	16.63	24.25	2.14
p2106*	1,262	3,102	2,424	1,523+42	560+13	t	t	t	t	t/m	t/m	t	17.22	t	t	t	36.45	m	393.41	10.85	5.11
integrator_arm9*	1,267	50,606	3,024	1,599+12	550+5	t	t	t/m	t/m	t	t	t	53.74	t	t	t	263.43	t/m	29.86	32.75	6.69
olpce2294*	1,274	3,881	2,525	1,587+10	572+4	t	t	t	t	t	t	t/m	t/m	t	t	t	84.96	t/m	585.58	m	7.13
adderii*	1,276	3,206	2,522	1,581+12	591+5	t	t	t	t	t	t	t/m	33.22	t/m	t/m	t	t	t/m	16.56	m	10.69
at91sam7sek*	1,296	3,921	2,530	1,600+12	571+5	t	t	t	t	t/m	t/m	t	45.75	t	127.96	t	17.60	63.58	25.59	15.24	3.96
se77x9*	1,319	49,937	3,027	1,639+7	577+3	t	t	t	t	t/m	t/m	t/m	t/m	t	t	t	52.07	m	25.78	60.59	4.96
m5272c3*	1,323	3,297	2,598	1,636+11	627+4	t	t	t	t	t/m	t/m	t/m	56.59	t	t	t	98.02	t/m	t/m	39.66	6.07
phycore229x*	1,360	4,026	2,622	1,677+10	578+4	t	t	t	t	t/m	t/m	t	73.57	t/m	578.17	t	20.27	m	22.76	97.27	2.58
freebsd-icse11	1,396	62,183	14,094	1,563+1	7	t	t	t	t	t	t	t	t	t/m	t/m	t/m	t/m	t/m	t/m	t	t
ea2468*	1,408	3,470	2,649	1,680+38	612+10	t	t	t	t	t/m	t/m	t/m	53.57	t/m	t/m	t	45.46	t/m	61.07	26.37	2.36
uclinux-distribution	1,580	2,131	1,389	1,146+3	49+1	t/m	t/m	t/m	t/m	m	m	m	m	0.01	0.01	0.01	0.00	0.01	0.00	0.00	0.00
fiasco	1,638	5,228	3,899	2,052+50	137+21	t	t	t	22.63	t/m	t/m	t/m	15.01	t/m	t/m	t/m	32.71	0.38	0.14	17.14	0.69
uclinux	1,850	2,468	1,850	1,547	303	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
automotive01	2,513	10,275	9,845	4,398+27	430+12	t/m	t/m	t	t	t/m	t/m	t/m	3.95	t	23.24	24.26	3.14	1.07	0.18	0.83	0.25
linux-2.6.33.3	6,467	132,032	91,686	8,290+28	359+7	m	m	t/m	t/m	m	m	t/m	t/m	m	m	t/m	t/m	t/m	t/m	t/m	t/m
busybox-1.18.0-smarch	6,796	17,836	16,044	7,143+183	969+71	t/m	t/m	t/m	t/m	t/m	t/m	t/m	t/m	t	t	0.06	0.04	0.33	0.10	0.04	0.02
uclinux-config	11,254	31,637	29,955	11,761+324	2,156+109	t/m	t/m	m	m	t/m	t/m	m	m	t	t	0.16	0.07	m	1.14	0.06	0.04
buildroot	14,910	45,603	44,006	18,062+147	4,290+70	t/m	t/m	t/m	t/m	t/m	t/m	m	m	t/m	t/m	t/m	t/m	m	m	t/m	t/m
automotive2_4	18,616	350,221	333,424	6,108+18	1,483+9	m	m	t	t	m	m	t/m	t/m	m	m	0.70	0.29	t/m	8.10	0.22	0.14
embtoolkit-smarch	23,516	180,511	157,717	30,380+1,818	7,316+176	m	m	t/m	t/m	m	m	m	m	m	m	m	m	t/m	t/m	t/m	t/m
freetz	31,012	102,705	99,356	35,143+2,607	7,113+880	m	m	t/m	t/m	m	m	m	m	m	m	m	m	m	m	t/m	t/m

constructed (po-rc). Logic2BDD's dynamic reordering enhances all base configurations except po-r, with a remarkable benefit for ∅-f. Still, it remains the worst-performing base configuration. The performance decline for po-r is possibly due to the inherent costs of dynamic reordering in terms of compilation time, which do not necessarily pay off in a substantial BDD node count reduction. Another aspect could be that the sifting-based reordering algorithm is limited to a local view on the problem. While static ordering heuristics look at the entire XCNF, sifting can only take the clauses into account that have been compiled into the BDD so far, and thus may counter the good initial variable order [22]. Compared to previous experiments without ONE-HOT factorization [21], however, we barely observe these effects, so apparently ONE-HOT factorization mostly mitigates the potentially negative impact of dynamic variable reordering.

Throughout all configurations and engines, the same 16 models always finished within two seconds, which we label as *easy*. For these models, the individual influences of the configuration options were mixed, and negligible given the short construction times. For the remaining *non-easy* models, a clear impact of the configurations is visible: the highest number of models can be constructed for the combined configuration po-rc, and gradually less when dropping either of the configurations or switching from ReMINCE to FORCE (see Figure 10).

Table 3 shows detailed timings for OxiDD with a balanced construction scheme, showing for each model



(a) with left-deep construction scheme

(b) with balanced construction scheme

Figure 11: Parallelization speedups for UnWise models with `po-rc` preprocessing and construction using OxiDD. There is one point per run, i.e., no aggregation over the five seeds.

the best (min) and worst (max) construction time across the five random seeds. While showing mixed influences of the configuration options, also here the use of PMC preprocessing and ONE-HOT factorization in combination with ReMINCE variable and clause ordering shows the best performance (`po-rc-ob`). The best-performing runs with configuration `po-rc-ob` enabled to construct 44 out of the 49 models of the benchmark set in each at most 11 seconds.

Towards answering **RQ1a-c**, we find that a balanced OxiDD construction with ReMINCE variable- and clause-ordering and PMC preprocessing with ONE-HOT factorization performs best. It clearly outperforms Logic2BDD, the current state-of-the-art, which only performs well on small feature models.

4.3.4. Parallelization and Construction Schemes (E4)

The huge improvement of a balanced construction (`ob`) over the left-deep scheme (`ol`) deserves a closer look. To this end, we focus on the best preprocessing configuration, i.e., `po-rc`. Besides balanced and left-deep, OxiDD offers the work-stealing scheme, which requires parallelization. To compare all three schemes, we first investigate the impact of parallelization.

Figure 11 shows considerable speedups for models whose single-threaded construction takes at least 0.001 s. The speedups for the balanced scheme tend to be a bit higher, which could be due to the concurrent execution of multiple `apply_and` operations. But even for larger models, the speedup barely reaches a factor 4 for four threads, so the scalability of parallelization appears to be limited.

As an exemplary model, we consider `at91sam7sek` from the UnWise test suite. In single-threaded execution, *left-deep* constructions takes 72.8 s, while for the *balanced* tree, we only need 5.7 s. When switching to multi-threaded construction using 4 worker threads, we observed running times of 39.2 s (*left-deep*) and 3.7 s (*balanced*). Allowing to reassociate operations (*work-stealing*) is not beneficial compared to the balanced tree, the construction time is 4.0 s here.

To investigate the reason for these large differences, we plot the descendant count $|n|$ for the result node n of each conjunction operation in Figure 12c (left). Note that the final count of processed conjunctions is less than the clause count because OxiDD-CLI has special handling for unit clauses. For the *left-deep* tree, there are considerably more large conjuncts, as can be seen in the histogram (Figure 12c, right). The graph shape for the *balanced* tree follows our expectations: in the first half of processed conjunctions, we only combine clauses from the input, of which all consist of at most five literals. In the worst case, the result of `apply_and`(n_1, n_2) has at most $|n_1| \cdot |n_2|$ descendant nodes. Therefore, the first half of conjuncts is guaranteed to be small again. Results keep growing while moving up in the tree. The pattern for *work-stealing* is similar.

The pattern of conjunct sizes for the `at91sam7sek` model is quite typical. Together with the other five patterns shown in Figure 12 it is representative for the UnWise suite with `po-rc` preprocessing. We remark that cases as in Figure 12d are rare, where intermediate results grow much larger than the final BDD size indicated by the red line. Apparently, even in left-deep constructions, the `po-rc` preprocessing configuration typically mitigates the well-known peak-size explosion problem [49]. This could be mainly due to effective clause ordering, which is shown to benefit from other preprocessing techniques [22].

Regarding **RQ1c**, we observe that the balanced and work-stealing schemes yield smaller intermediate BDDs, avoiding the peak-size explosion problem and greatly improving both construction time and memory usage. With respect to **RQ1d**, we find that parallelization may be beneficial, but its impact is more moderate.

4.4. Large-scale Models (RQ2)

The experiments on the UnWise benchmark set in Table 3 already showed promising results for large models from the eCos feature model collection [8]. In the literature,

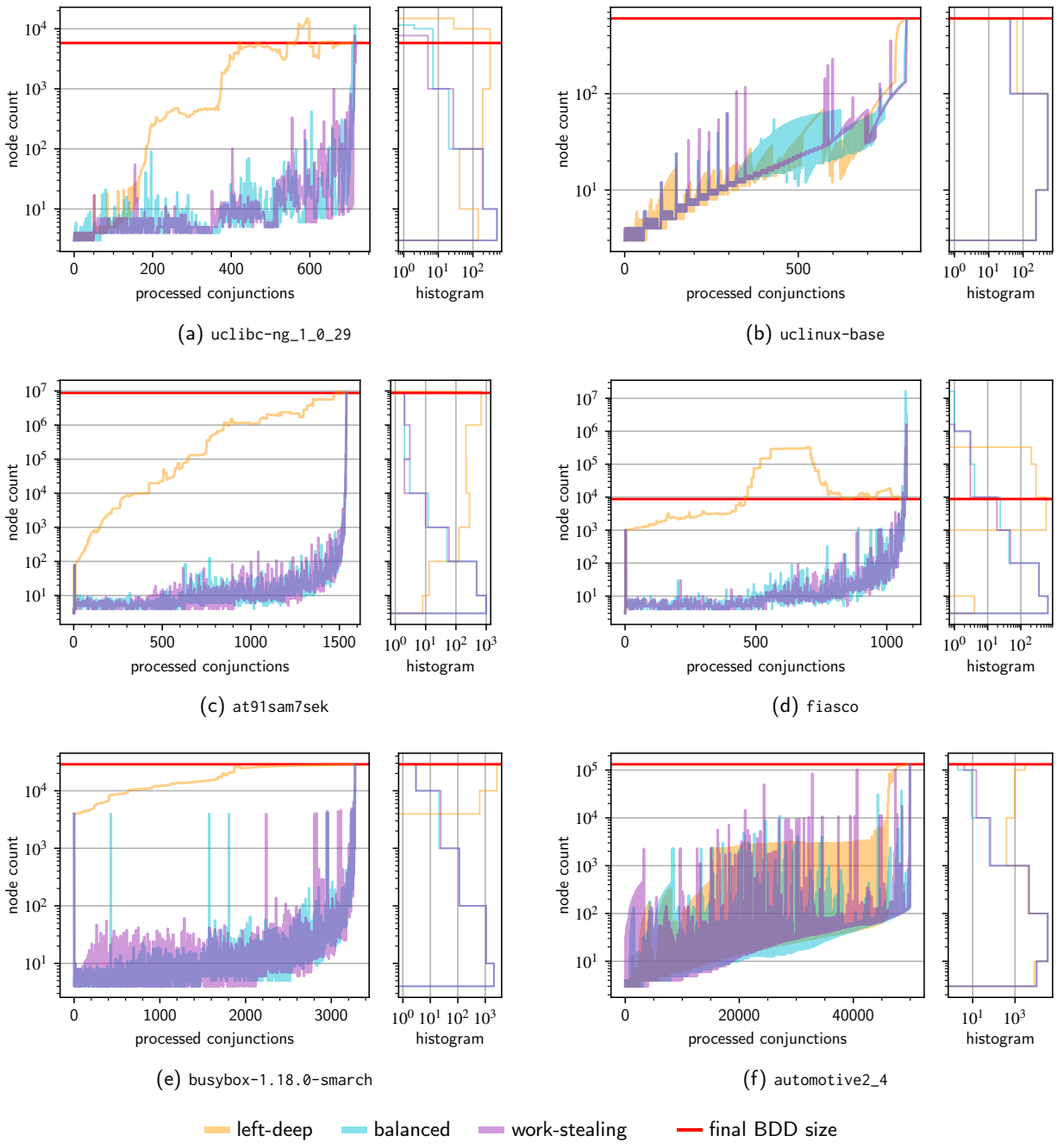
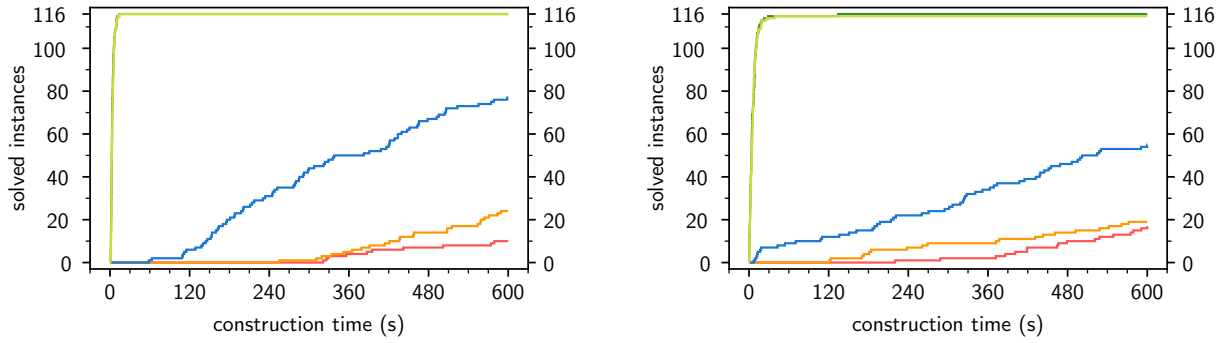


Figure 12: Construction scheme examples for BDD sizes after performing conjunctions.

these models were considered too complex for BDD compilation [28]. While the authors of UnWise randomly selected 12 out of the 116 eCos models for their benchmark set, we now consider the whole collection. For this, we generated the 116 CNF input models from feature models given as feature diagrams [38] using FeatureIDE v3.11.1 [60]. Following the results of Section 4.3, we executed each experiment five times with the configuration `po-rc` that showed the best performance in our previous experiments. Contributing to **RQ1a**, we also compare `po-rc` to the configuration `p-rc` to study the impact of ONE-HOT factorization on large-scale models. Figure 13 shows the number of models successfully constructed in at least one run for different BDD engines and construction schemes. In the experiments, OxiDD with parallel balanced or work-stealing construction (`opb`/`opw`) clearly outperforms all other approaches. With ONE-HOT factorization enabled, these two configurations allow to construct all eCos models in less than 13.7 s or 14.3 s each, respectively. We hence could construct all eCos models in a total time of 426 s, i.e., around 7 minutes. Only about two thirds of the models can be constructed using OxiDD left-deep (`opl`). Logic2BDD performs best without



(a) with ONE-HOT factorization (`po-rc`)

(b) without ONE-HOT factorization (`p-rc`)

	Solved Instances			
	with factorization (`po-rc`)		without factorization (`p-rc`)	
Configuration	#	max time (s)	#	max time (s)
OxiDD balanced (`opb`)	116	13.7	116	134.2
OxiDD work-stealing (`opw`)	116	14.3	115	39.0
OxiDD left-deep (`op1`)	77	598.5	55	599.9
Logic2BDD no reordering (`21`)	24	589.0	19	572.0
Logic2BDD reordering (`2d1`)	10	579.0	17	600.0

Figure 13: Solved instances of the large models (eCos) with and without ONE-HOT factorization. Timings are for the last successfully compiled model of the minimum over five runs with random seeds, i.e., disregarding timeouts [>600 s].

dynamic variable reordering (`21`), which can be explained by this optimization changing a potentially good initial variable order gained through ReMINCE to a worse one [22] (see Section 4.3.3). But even with the best configuration of Logic2BDD, only 20 % of the models could be compiled within 10 minutes.

We observe that OxiDD and Logic2BDD configurations without reordering can greatly benefit from ONE-HOT factorization. However, for left-deep constructions, some models with short compilation times (less than 2 minutes for OxiDD and less than 5 minutes for Logic2BDD) are slowed down. Interestingly, ONE-HOT factorization is also not beneficial for Logic2BDD with reordering enabled, most likely due to specific techniques implemented in Logic2BDD that change the variable order for ONE-HOT groups. This also shows that for large-scale feature models, a good initial variable order obtained by suitable static variable-ordering heuristics is more important than dynamic reordering techniques, which are in turn mostly effective for small feature models (also see Figure 10d).

Regarding **RQ2**, we conclude that optimizing BDD compilation configurations drastically improves scalability and can enable the construction of large-scale feature models such as all eCos feature models.

While our techniques enabled the construction of a well-known collection of large-scale feature models, we did not yet succeed to scale construction to the most complex models from the UnWise test suite (see Table 3).[9]

4.5. Threats to Validity

Internal Validity. Computing variable and clause orders with MINCE, ReMINCE, and FORCE showed fluctuating performance, due to the probabilistic nature of hypergraph cutting in MINCE and ReMINCE and the dependence on the initial order for FORCE. To estimate and mitigate this impact, we ran each experiment five times with five different random seeds to shuffle the initial variable and clause order. More runs would, however, only further improve the performance, and the differences in the configurations were mostly clear in all five executions. We hence consider this threat low. Statistical tests would strengthen evidence but require even more experiments. The aim of the paper was to compare the large configuration space of CNF-to-BDD compilation, not statistical considerations on single techniques. The conversion of feature models to CNF could also influence our results, where we followed the standard approach of generating DIMACS files via FeatureIDE [60] from XML feature models. We implemented several sanity checks to ensure that applied model transformations preserve semantic equivalence, e.g., by comparing model counts (#SAT).

[9] The models `linux-2.6.33.3`, `freebsd-icse11`, `buildroot`, `freetz`, and `embtoolkit-smarch` could not be constructed in any of the test runs.

External Validity. Our benchmarks may not be representative for all kinds of feature models. However, we relied on state-of-the-art and established test suites covering a wide range of applications, including the recent selection of feature models for the sampler UnWise [26] and the standard eCos benchmarks [8, 38], where validity has been confirmed already in previous publications. Further, other BDD compilation and CNF preprocessing techniques from the literature threat validity. Here, our focus was not to provide an exhaustive comparison of existing techniques, but conduct a first comparison of techniques in CNF-to-BDD compilation and SAT preprocessing. Also, different BDD tools other than OxiDD and Logic2BDD could be included for a comparison. However, our goal was not to compare BDD libraries but the effectiveness of techniques for feature model compilation. Towards a comparison of BDD libraries and CNF-to-BDD compilation in domains other than feature-oriented modeling, we refer to [32].

5. Conclusion

To the best of our knowledge, we were the first who conducted a structured impact analysis of CNF-to-BDD compilation techniques for feature models. Our analysis included PMC preprocessing, ONE-HOT and XOR factorization, various variable- and clause-ordering heuristics, as well as different construction schemes and parallelization. We showed that configuring BDD compilation greatly improves performance or even enables construction of large-scale feature models. The latter we witnessed by all 116 eCos models together being constructible in around 7 minutes, where existing approaches without optimizing BDD compilation configurations could not succeed even in any single one.

To summarize, our findings include:

- CNF-to-BDD compilation of feature models performs best when done with PMC preprocessing and ONE-HOT factorization, ReMINCE variable and clause ordering, as well as multi-threaded balanced construction using OxiDD.
- Balanced construction schemes greatly benefit from PMC preprocessing, ONE-HOT factorization, and ReMINCE clause ordering.
- Dynamic variable reordering is only beneficial for smaller feature models but worsens performance for larger ones.

Acknowledgments

The authors would like to thank the anonymous reviewers for their comments that helped to improve the quality of the manuscript. This work was partially supported by the DFG under the projects TRR 248 (see https://perspicuous-computing.science, project ID 389792660) and EXC 2050/1 (CeTI, project ID 390696704, as part of Germany's Excellence Strategy) and by the NWO through Veni grant VI.Veni.222.431.

References

[1] Akers, S.B., 1978. Binary decision diagrams. IEEE Transactions Computers 27, 509–516. doi:10.1109/TC.1978.1675141.

[2] Aloul, F.A., 2003. Mince. URL: http://www.aloul.net/Tools/mince/.

[3] Aloul, F.A., Markov, I.L., Sakallah, K.A., 2003. FORCE: A fast and easy-to-implement variable-ordering heuristic, in: GLSVLSI, ACM. pp. 116–119. doi:10.1145/764808.764839.

[4] Aloul, F.A., Markov, I.L., Sakallah, K.A., 2004. MINCE: A static global variable-ordering heuristic for SAT search and BDD manipulation., in: JUCS, pp. 1562–1596. doi:10.3217/jucs-010-12-1562.

[5] Apel, S., Batory, D., Kästner, C., Saake, G., 2013. Feature-oriented software product lines. Springer. doi:10.1007/978-3-642-37521-7.

[6] Batory, D., 2005. Feature models, grammars, and propositional formulas, in: Obbink, H., Pohl, K. (Eds.), Software Product Lines, Springer, Berlin, Heidelberg. pp. 7–20. doi:10.1007/11554844_3.

[7] Benavides, D., Segura, S., Ruiz-Cortés, A., 2010. Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35, 615–636. doi:10.1016/j.is.2010.01.001.

[8] Berger, T., She, S., Lotufo, R., Wąsowski, A., Czarnecki, K., 2010. Variability modeling in the real: a perspective from the operating systems domain, in: ASE, ACM. pp. 73–82. doi:10.1145/1858996.1859010.

[9] Biere, A., Heule, M., van Maaren, H., Walsh, T. (Eds.), 2021. Handbook of Satisfiability - Second Edition. volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. doi:10.3233/FAIA336.

[10] Blumofe, R.D., Leiserson, C.E., 1999. Scheduling multithreaded computations by work stealing, in: J. ACM, pp. 720–748. doi:10.1145/324133.324234.

[11] Bollig, B., Wegener, I., 1996. Improving the variable ordering of OBDDs is NP-complete. IEEE Transactions on Computers 45, 993–1002. doi:10.1109/12.537122.

[12] Brace, K., Rudell, R., Bryant, R., 1990. Efficient implementation of a BDD package, in: DAC, pp. 40–45. doi:10.1109/DAC.1990.114826.

[13] Bryant, R.E., 1992. Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys 24, 293–318. doi:10.1145/136035.136043.

[14] Cadoli, M., Donini, F.M., 1997. A survey on knowledge compilation. AI Commun. 10, 137–150. URL: https://api.semanticscholar.org/CorpusID:16746005.

[15] Caldwell, A., Kahng, A., Markov, I., 2000. Can recursive bisection alone produce routable, placements?, in: DAC, pp. 477–482. doi:10.1145/337292.337549.

[16] de Colnet, A., 2023. Separating Incremental and Non-Incremental Bottom-Up Compilation, in: Mahajan, M., Slivovsky, F. (Eds.), SAT, Schloss Dagstuhl – LZI, Dagstuhl, Germany. pp. 7:1–7:20. doi:10.4230/LIPIcs.SAT.2023.7.

[17] Darras, S., Dequen, G., Devendeville, L., Mazure, B., Ostrowski, R., Saïs, L., 2005. Using Boolean constraint propagation for sub-clauses deduction, in: van Beek, P. (Ed.), CP, Springer, Berlin, Heidelberg. pp. 757–761. doi:10.1007/11564751_59.

[18] Darwiche, A., 2002. A compiler for deterministic, decomposable negation normal form, in: AAAI, AAAI Press, USA. pp. 627–634. doi:10.5555/777092.777189.

[19] Darwiche, A., Marquis, P., 2002. A knowledge compilation map. JAIR 17, 229–264. doi:10.1613/JAIR.989.

[20] van Dijk, T., van de Pol, J., 2017. Sylvan: multi-core framework for decision diagrams. STTT 19, 675–696. doi:10.1007/s10009-016-0433-2.

[21] Dubslaff, C., Husung, N., Käfer, N., 2024. Configuring BDD compilation techniques for feature models, in: Proceedings of the 28th ACM International Systems and Software Product Line Conference, ACM, New York, NY, USA. pp. 209–216. doi:10.1145/3646548.3676538.

[22] Dubslaff, C., Wirtz, J., 2025. Compiling Binary Decision Diagrams with Interrupt-Based Downsizing. Springer Nature Switzerland, Cham. pp. 252–273. doi:10.1007/978-3-031-75778-5_12.

[23] Fernandez-Amoros, D., Bra, S., Aranda-Escolástico, E., Heradio, R., 2020. Using extended logical primitives for efficient BDD building. Mathematics 8. doi:10.3390/math8081253.

[24] Fortune, S., Hopcroft, J., Schmidt, E.M., 1978. The complexity of equivalence and containment for free single variable program schemes, in: Ausiello, G., Böhm, C. (Eds.), ICALP, Springer, Berlin, Heidelberg. pp. 227–240. doi:10.1007/3-540-08860-1_17.

[25] Heß, T., Müller, T., Sundermann, C., Thüm, T., 2022. ddueruem: a wrapper for feature-model analysis tools, in: SPLC, ACM, New York, NY, USA. pp. 54–57. doi:10.1145/3503229.3547032.

[26] Heß, T., Schmidt, T.J., Ostheimer, L., Krieter, S., Thüm, T., 2024. UnWise: High T-Wise Coverage from Uniform Sampling, in: VaMoS, ACM, New York, NY, USA. pp. 37–45. doi:10.1145/3634713.3634716.

[27] Heß, T., Semmler, S.N., Sundermann, C., Torán, J., Thüm, T., 2024. Towards deterministic compilation of binary decision diagrams from feature models, in: SPLC, ACM, New York, NY, USA. pp. 136–147. doi:10.1145/3646548.3672598.

[28] Heß, T., Sundermann, C., Thüm, T., 2021. On the scalability of building binary decision diagrams for current feature models, in: SPLC, ACM, New York, NY, USA. pp. 131–135. doi:10.1145/3461001.3474452.

[29] Heule, M.J.H., 2008. SmArT solving: tools and techniques for satisfiability solvers. Ph.D. thesis. Delft University of Technology, Netherlands. URL: http://resolver.tudelft.nl/uuid:d41522e3-690a-4eb7-a352-652d39d7ac81.

[30] Huang, J., Darwiche, A., 2004. Using DPLL for efficient OBDD construction, in: Hoos, H.H., Mitchell, D.G. (Eds.), SAT, Springer, Berlin, Heidelberg. pp. 157–172. doi:10.1007/11527695_13.

[31] Husung, N., Dubslaff, C., Hermanns, H., Köhl, M.A., 2024. OxiDD: A safe, concurrent, modular, and performant decision diagram framework in Rust, in: TACAS, Springer. doi:10.1007/978-3-031-57256-2_13.

[32] Husung, N., Dubslaff, C., Hermanns, H., Köhl, M.A., 2025a. OxiDD: The next-gen decision diagram framework in Rust, in: International Journal on Software Tools for Technology Transfer (STTT). Accepted for publication.

[33] Husung, N., Käfer, N., Dubslaff, C., 2025b. Dimagic 2.0. doi:10.5281/zenodo.15583680.

[34] Jain, J., Narayan, A., Coelho, C., Khatri, S.P., Sangiovanni-Vincentelli, A., Brayton, R.K., Fujita, M., 1995. Combining Top-down and Bottom-up approaches for ROBDD. Technical Report. University of California at Berkeley. URL: https://digicoll.lib.berkeley.edu/record/135872/files/ERL-95-30.pdf.

[35] Käfer, N., Apel, S., Baier, C., Dubslaff, C., Hermanns, H., 2025. When to sample from feature diagrams?, in: VaMoS, ACM, New York, NY, USA. pp. 11–20. doi:10.1145/3715340.3715442.

[36] Käfer, N., Husung, N., Dubslaff, C., 2024. dimagic. doi:10.5281/zenodo.12707100.

[37] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report. Carnegie-Mellon University, Software Engineering Institute. URL: https://www.sei.cmu.edu/documents/1011/1990_005_001_15872.pdf.

[38] Knüppel, A., Thüm, T., Mennicke, S., Meinicke, J., Schaefer, I., 2017. Is there a mismatch between real-world feature models and product-line research?, in: ESEC/FSE, ACM, New York, NY, USA. pp. 291–302. doi:10.1145/3106237.3106252.

[39] Knuth, D.E., 2009. The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams. 12th ed., Addison-Wesley Professional.

[40] Korhonen, T., Järvisalo, M., 2021. Integrating Tree Decompositions into Decision Heuristics of Propositional Model Counters, in: Michel, L.D. (Ed.), CP, Schloss Dagstuhl – LZI, Dagstuhl, Germany. pp. 8:1–8:11. doi:10.4230/LIPIcs.CP.2021.8.

[41] Lagniez, J.M., Marquis, P., 2014. Preprocessing for propositional model counting, in: AAAI, AAAI Press. pp. 2688–2694. doi:10.1609/AAAI.V28I1.9116.

[42] Lind-Nielsen, J., 2004. BuDDy: A binary decision diagram package, version 2.4. URL: https://buddy.sourceforge.net/manual/.

[43] Marques-Silva, J., Janota, M., Lynce, I., 2010. On computing backbones of propositional theories, in: ECAI, IOS Press, NLD. pp. 15–20. doi:10.3233/978-1-60750-606-5-15.

[44] Mendonca, M., Wasowski, A., Czarnecki, K., Cowan, D., 2008. Efficient compilation techniques for large scale feature models, in: GPCE, ACM, New York, NY, USA. pp. 13–22. doi:10.1145/1449913.1449918.

[45] Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., Troyansky, L., 1999. Determining computational complexity from characteristic 'phase transitions'. Nature 400, 133–137. doi:10.1038/22055.

[46] Mrena, M., Kvassay, M., 2021. Comparison of left fold and tree fold strategies in creation of binary decision diagrams, in: IDT, pp. 341–352. doi:10.1109/IDT52577.2021.9497593.

[47] Piette, C., Hamadi, Y., Sais, L., 2008. Vivifying propositional clausal formulae, in: Ghallab, M., Spyropoulos, C.D., Fakotakis, N., Avouris, N.M. (Eds.), ECAI, IOS Press. pp. 525–529. doi:10.3233/978-1-58603-891-5-525.

[48] Popov, M., Balyo, T., Iser, M., Ostertag, T., 2023. Construction of decision diagrams for product configuration, in: Horcas, J.M., Galindo, J.A., Comploi-Taupe, R., Fuentes, L. (Eds.), ConfWS, CEUR-WS.org. pp. 108–117. URL: https://ceur-ws.org/Vol-3509/paper15.pdf.

[49] Qayyum, K., Mahzoon, A., Drechsler, R., 2022. Monitoring the effects of static variable orders on the construction of BDDs, in: MESIICON, pp. 1–6. doi:10.1109/MESIICON55227.2022.10093493.

[50] Rice, M., Kulhari, S., 2008. A survey of static variable ordering heuristics for efficient BDD/MDD construction. Technical Report. University of California. URL: http://alumni.cs.ucr.edu/~skulhari/StaticHeuristics.pdf.

[51] Roy, J.A., Markov, I.L., Bertacco, V., 2004. Restoring circuit structure from SAT instances, in: IWLS. URL: https://www.academia.edu/download/30670628/10.1.1.2.9660.pdf.

[52] Rudell, R., 1993. Dynamic variable ordering for ordered binary decision diagrams, in: ICCAD, pp. 42–47. doi:10.1109/ICCAD.1993.580029.

[53] Schlag, S., Heuer, T., Gottesbüren, L., Akhremtsev, Y., Schulz, C., Sanders, P., 2022. High-quality hypergraph partitioning, in: ACM J. Exp. Algorithmics. doi:10.1145/3529090.

[54] She, S., Lotufo, R., Berger, T., Wąsowski, A., Czarnecki, K., 2010. Variability model of the Linux kernel, in: VaMoS, Linz, Austria. URL: http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf.

[55] Shmoys, D.B., 1997. Cut problems and their application to divide-and-conquer, in: Approximation algorithms for NP-hard problems, pp. 192–235.

[56] Somenzi, F., 2015. CUDD: CU Decision Diagram Package. Technical Report. University of Colorado at Boulder.

[57] Soos, M., Meel, K.S., 2019. BIRD: Engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting, in: AAAI, pp. 1592–1599. doi:10.1609/aaai.v33i01.33011592.

[58] Sundermann, C., Kuiter, E., Heß, T., Raab, H., Krieter, S., Thüm, T., 2023. On the benefits of knowledge compilation for feature-model analyses. Annals of Mathematics and Artificial Intelligence doi:10.1007/s10472-023-09906-6.

[59] Thüm, T., 2020. A BDD for linux? the knowledge compilation challenge for variability, in: SPLC, ACM, New York, NY, USA. doi:10.1145/3382025.3414943.

[60] Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T., 2014. FeatureIDE: An extensible framework for feature-oriented software development. SCP 79, 70–85. doi:10.1016/j.scico.2012.06.002.

[61] Zhang, Y., Zhao, Z., Chen, G., Song, F., Chen, T., 2023. Precise quantitative analysis of binarized neural networks: A BDD-based approach. TOSEM 32. doi:10.1145/3563212.