A Top-Down Compiler for Sentential Decision Diagrams

Umut Oztok and **Adnan Darwiche**
Computer Science Department
University of California, Los Angeles
{umut,darwiche}@cs.ucla.edu

Abstract

The sentential decision diagram (SDD) has been recently proposed as a new tractable representation of Boolean functions that generalizes the influential ordered binary decision diagram (OBDD). Empirically, compiling CNFs into SDDs has yielded significant improvements in both time and space over compiling them into OBDDs, using a *bottom-up* compilation approach. In this work, we present a *top-down* CNF to SDD compiler that is based on techniques from the SAT literature. We compare the presented compiler empirically to the state-of-the-art, bottom-up SDD compiler, showing orders-of-magnitude improvements in compilation time.

1 Introduction

The area of *knowledge compilation* has a long tradition in AI (see, e.g., Marquis [1995], Selman and Kautz [1996], Cadoli and Donini [1997]). Since Darwiche and Marquis [2002], this area has settled on three major research directions: (1) identifying new tractable representations that are characterized by their succinctness and polytime support for certain queries and transformations; (2) developing efficient knowledge compilers; and (3) using those representations and compilers in various applications, such as diagnosis [Barrett, 2005; Elliott and Williams, 2006], planning [Barrett, 2004; Palacios *et al.*, 2005; Huang, 2006], and probabilistic inference [Chavira and Darwiche, 2008]. For a recent survey on knowledge compilation, see Darwiche [2014].

This work focuses on developing efficient compilers. In particular, our emphasis is on the compilation of the sentential decision diagram (SDD) [Darwiche, 2011] that was recently discovered as a tractable representation of Boolean functions. SDDs are a strict superset of ordered binary decision diagrams (OBDDs) [Bryant, 1986], which are one of the most popular, tractable representations of Boolean functions. Despite their generality, SDDs still maintain some key properties behind the success of OBDDs in practice. This includes canonicity, which leads to unique representations of Boolean functions. It also includes the support of an efficient `Apply` operation that combines SDDs using Boolean operators.[1] SDDs also come with tighter size upper bounds than OBDDs [Darwiche, 2011; Oztok and Darwiche, 2014; Razgon, 2014b]. Moreover, SDDs have been used in different applications, such as probabilistic planning [Herrmann and de Barros, 2013], probabilistic logic programs [Vlasselaer *et al.*, 2014; 2015], probabilistic inference [Choi *et al.*, 2013; Renkens *et al.*, 2014], verification of multi-agent systems [Lomuscio and Paquet, 2015], and tractable learning [Kisa *et al.*, 2014; Choi *et al.*, 2015].

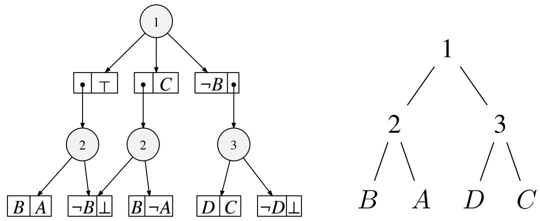
Almost all of these applications are based on the *bottom-up* SDD compiler developed by Choi and Darwiche [2013a], which was also used to compile CNFs into SDDs [Choi and Darwiche, 2013b]. This compiler constructs SDDs by compiling small pieces of a knowledge base (KB) (e.g., clauses of a CNF). It then combines these compilations using the `Apply` operation to build a compilation for the full KB.

An alternative to bottom-up compilation is *top-down* compilation. This approach starts the compilation process with a full KB. It then recursively compiles the fragments of the KB that are obtained through conditioning. The resulting compilations are then combined to obtain the compilation of the full KB. All existing top-down compilers assume CNFs as input, while bottom-up compilers can work on any input due to the `Apply` operation. Yet, compared to bottom-up compilation, top-down compilation has been previously shown to yield significant improvements in compilation time and space when compiling CNFs into OBDDs [Huang and Darwiche, 2004]. Thus, it has a potential to further improve the results on CNF to SDD compilations. Motivated by this, we study the compilation of CNFs into SDDs by a top-down approach.

This paper is based on the following contributions. We first identify a subset of SDDs, called *Decision-SDDs*, which facilitates the top-down compilation of SDDs. We then introduce a top-down algorithm for compiling CNFs into Decision-SDDs, which is harnessed with techniques used in modern SAT solvers, and a new component caching scheme. We finally present empirical results, showing orders-of-magnitude improvement in compilation time, compared to the state-of-the-art, bottom-up SDD compiler.

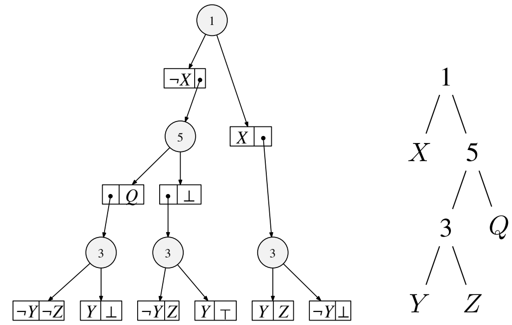
This paper is organized as follows. Section 2 provides technical background. Section 3 introduces the new representa-

[1] The `Apply` operation combines two SDDs using any Boolean operator, and has its origins in the OBDD literature [Bryant, 1986].



(a) An SDD (b) A vtree

Figure 1: An SDD and a vtree for $(A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$.



(a) A Decision-SDD (b) A vtree

Figure 2: A Decision-SDD and its corresponding vtree.

tion Decision-SDD. Section 4 provides a formal framework for the compiler, which is then presented in Section 5. Experimental results are given in Section 6 and related work in Section 7. Due to space limitations, proofs of theorems are delegated to the full version of the paper.[2]

2 Technical Background

Upper case letters (e.g., X) denote variables and bold upper case letters (e.g., $\mathbf{X}$) denote sets of variables. A *literal* is a variable or its negation. A *Boolean function* $f(\mathbf{Z})$ maps each instantiation $\mathbf{z}$ of variables $\mathbf{Z}$ to true ($\top$) or false ($\bot$).

CNF: A *conjunctive normal form* (CNF) is a set of clauses, where each clause is a disjunction of literals. Conditioning a CNF Δ on a literal ℓ, denoted $\Delta|\ell$, amounts to removing literal $\neg\ell$ from all clauses and then dropping all clauses that contain literal ℓ. Given two CNFs Δ and Γ, we will write $\Delta \models \Gamma$ to mean that Δ entails Γ.

SDD: A Boolean function $f(\mathbf{X}, \mathbf{Y})$, with disjoint sets of variables $\mathbf{X}$ and $\mathbf{Y}$, can always be decomposed into

$$f(\mathbf{X}, \mathbf{Y}) = \big(p_1(\mathbf{X}) \wedge s_1(\mathbf{Y})\big) \vee \ldots \vee \big(p_n(\mathbf{X}) \wedge s_n(\mathbf{Y})\big),$$

such that $p_i \neq \bot$ for all i; $p_i \wedge p_j = \bot$ for $i \neq j$; and $\bigvee_i p_i = \top$. A decomposition satisfying the above properties is known as an $(\mathbf{X}, \mathbf{Y})$-partition [Darwiche, 2011]. Moreover, each p_i is called a *prime,* each s_i is called a *sub*, and the $(\mathbf{X}, \mathbf{Y})$-partition is said to be *compressed* when its subs are distinct, i.e., $s_i \neq s_j$ for $i \neq j$.

SDDs result from the recursive decomposition of a Boolean function using $(\mathbf{X}, \mathbf{Y})$-partitions. To determine the $\mathbf{X}/\mathbf{Y}$ variables of each partition, we use a *vtree*, which is a full binary tree whose leaves are labeled with variables; see Figure 1(b). Consider now the vtree in Figure1(b), and also the Boolean function $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$. Node $v = 1$ is the vtree root. Its left subtree contains variables $\mathbf{X} = \{A, B\}$ and its right subtree contains $\mathbf{Y} = \{C, D\}$. Decomposing function f at node $v = 1$ amounts to generating an $(\mathbf{X}, \mathbf{Y})$-partition:

$$\{(\underbrace{A \wedge B}_{\text{prime}}, \underbrace{\top}_{\text{sub}}), (\underbrace{\neg A \wedge B}_{\text{prime}}, \underbrace{C}_{\text{sub}}), (\underbrace{\neg B}_{\text{prime}}, \underbrace{D \wedge C}_{\text{sub}})\}.$$

This partition is represented by the root node of Figure 1(a). This node, which is a circle, represents a *decision node* with three branches, where each branch is called an *element*. Each element is depicted by a paired box $\boxed{p \mid s}$. The left box corresponds to a prime p and the right box corresponds to its sub s. A prime p or sub s are either a constant, literal, or pointer to a decision node. In this case, the three primes are decomposed recursively, but using the vtree rooted at $v = 2$. Similarly, the subs are decomposed recursively, using the vtree rooted at $v = 3$. This decomposition process moves down one level in the vtree with each recursion, terminating at leaf vtree nodes.

SDDs constructed as above are said to *respect* the used vtree. These SDDs may contain trivial decision nodes which correspond to $(\mathbf{X}, \mathbf{Y})$-partitions of the form $\{(\top, \alpha)\}$ or $\{(\alpha, \top), (\neg\alpha, \bot)\}$. When these decision nodes are removed (by directing their parents to α), the resulting SDD is called *trimmed*. Moreover, an SDD is called *compressed* when each of its partitions is compressed. Compressed and trimmed SDDs are canonical for a given vtree [Darwiche, 2011]. Here, we restrict our attention to compressed and trimmed SDDs. Figure 1(a) depicts a compressed and trimmed SDD for the above example. Finally, an SDD node representing an $(\mathbf{X}, \mathbf{Y})$-partition is *normalized* for the vtree node v with variables $\mathbf{X}$ in its left subtree v^l and variables $\mathbf{Y}$ in its right subtree v^r. In Figure 1(a), SDD nodes are labeled with vtree nodes they are normalized for.

3 Decision-SDDs

We will next define the language of *Decision-SDDs*, which is a strict subset of SDDs and a strict superset of OBDDs. Our new top-down compiler will construct Decision-SDDs.

To define Decision-SDDs, we first need to distinguish between internal vtree nodes as follows. An internal vtree node is a *Shannon* node if its left child is a leaf, otherwise it is a *decomposition* node. The variable labeling the left child of a Shannon node is called the *Shannon variable* of the node. Vtree nodes 1 and 3 in Figure 2(b) are Shannon nodes, with X and Y as their Shannon variables. Vtree node 5 is a decomposition node. An SDD node that is normalized for a Shannon (decomposition) vtree node is called a *Shannon (decomposition) decision node*. A Shannon decision node has the form $\{(X, \alpha), (\neg X, \beta)\}$, where X is a Shannon variable.

Definition 1 (Decision-SDD)**.** *A <u>Decision-SDD</u> is an SDD in which each decomposition decision node has the form* $\{(p, s_1), (\neg p, s_2)\}$ *where* $s_1 = \top$, $s_1 = \bot$, *or* $s_1 = \neg s_2$.

[2]Available at http://reasoning.cs.ucla.edu.

Figure 2 shows a Decision-SDD and a corresponding vtree for the CNF $\{Y \vee \neg Z, \neg X \vee Z, X \vee \neg Y, X \vee Q\}$. The language of Decision-SDDs is complete: every Boolean function can be represented by a Decision-SDD using an appropriate vtree.

For further insights into Decision-SDDs, note that a decomposition decision node must have the form $\{(f, g), (\neg f, \bot)\}$, $\{(f, \top), (\neg f, g)\}$, or $\{(f, \neg g), (\neg f, g)\}$. Moreover, these forms represent the Boolean functions $f \wedge g$, $f \vee g$, and $f \oplus g$, respectively, where f and g are over disjoint sets of variables.

If an SDD is based on a general vtree, it may or may not be a Decision-SDD. However, the following class of vtrees, identified by Oztok and Darwiche [2014], guarantees a Decision-SDD.

Definition 2 (Decision Vtree)**.** *A clause is compatible with an internal vtree node v iff the clause mentions some variables inside v^l and some variables inside v^r. A vtree for CNF Δ is said to be a decision vtree for Δ iff every clause in Δ is compatible with only Shannon vtree nodes.*[3]

Figure 2(b) depicts a decision vtree for the CNF $\{Y \vee \neg Z, \neg X \vee Z, X \vee \neg Y, X \vee Q\}$.

Proposition 1. *Let v be a decision vtree for CNF Δ. An SDD for Δ that respects vtree v must be a Decision-SDD.*

As such, the input to our compiler will be a CNF and a corresponding decision vtree, and the result will be a Decision-SDD for the CNF. Note that one can always construct a decision vtree for any CNF [Oztok and Darwiche, 2014].

When every internal vtree node is a Shannon vtree node (i.e., the vtree is right-linear), the Decision-SDD corresponds to an OBDD. A quasipolynomial separation between SDDs and OBDDs was given by Razgon [2014b]. As it turns out, the SDDs used to show this separation are actually Decision-SDDs. We now complement this result by showing that Decision-SDDs can be simulated by OBDDs with at most a quasipolynomial increase in size (it is currently unknown if this holds for general SDDs).

Theorem 1. *Every Decision-SDD with n variables and size N has an equivalent OBDD with size $\leq N^{1+\log n}$.*

The above result is based on [Razgon, 2014a], which simulates decomposable AND-OBDDs with OBDDs.[4]

Xue *et al.* [2012] have identified a class of Boolean functions f_i, with corresponding variable orders π_i, such that the OBDDs based on orders π_i have exponential size, yet the SDDs based on vtrees that dissect orders π_i have linear size.[5] Interestingly, the SDDs used in this result turn out to be Decision-SDDs as well. Hence, a variable order that blows up an OBDD can sometimes be dissected to obtain a vtree that leads to a compact Decision-SDD. This reveals the practical significance of Decision-SDDs despite the quasipolynomial simulation of Theorem 1. We finally note that there is no known separation result between Decision-SDDs and SDDs.

[3]Without loss of generality, Δ has no empty or unit clauses.

[4]A decomposable AND-OBDD can be turned into a Decision-SDD in polytime, but it is not clear whether the converse is true.

[5]A vtree dissects a variable order if the order is generated by a left-right traversal of the vtree.

Algorithm 1: $\texttt{SAT}(\Delta)$

Input: Δ : a CNF
Output: $\top$ if Δ is satisfiable; $\bot$ otherwise

1 $\Gamma \leftarrow \{\}$ `// learned clauses`
2 $D \leftarrow \langle\rangle$ `// decision sequence`
3 **while** true **do**
4   **if** *unit resolution detects a contradiction in* $\Delta \wedge \Gamma \wedge D$ **then**
5   **if** $D = \langle\rangle$ **then return** $\bot$
6     $\alpha \leftarrow$ asserting clause for (Δ, Γ, D)
7     $m \leftarrow$ the assertion level of α
8     $D \leftarrow$ the first m decisions of D
9     $\Gamma \leftarrow \Gamma \cup \{\alpha\}$ `// learning clause α`
10   **else**
11   **if** *ℓ is a literal where neither ℓ nor $\neg\ell$ are implied by unit resolution from $\Delta \wedge \Gamma \wedge D$* **then** $D \leftarrow D; \ell$
12   **else return** $\top$

We will next provide a top-down algorithm for compiling CNFs into Decision-SDDs, which is based on state-of-the-art techniques from SAT solving. Our intention is to provide a formal description of the algorithm, which is precise and detailed enough to be reproducible by the community. We will start by providing a formal description of our framework in Section 4, and then present our algorithm in Section 5.

4 A Formal Framework for the Compiler

Modern SAT solvers utilize two powerful and complementary techniques: *unit resolution* and *clause learning*. Unit resolution is an efficient, but incomplete, inference rule which identifies some of the literals implied by a CNF. Clause learning is a process which identifies clauses that are implied by a CNF, then adds them to the CNF so as to empower unit resolution (i.e., allows it to derive more literals). These clauses, also called *asserting clauses,* are learned when unit resolution detects a contradiction in the given CNF. We will neither justify asserting clauses, nor delve into the details of computing them, since these clauses have been well justified and extensively studied in the SAT literature (see, e.g., Moskewicz *et al.* [2001]). We will, however, employ asserting clauses in our SDD compiler (we employ first-UIP asserting clauses as implemented by RSat [Pipatsrisawat and Darwiche, 2007]).

As a first step towards introducing our compiler, we present in Algorithm 1 a modern SAT solver that is based on unit resolution and clause learning. This algorithm repeatedly performs the following process. A literal ℓ is chosen and added to the decision sequence D (we say that ℓ has been decided at *level* $|D|$). After deciding the literal ℓ, unit resolution is performed on $\Delta \wedge \Gamma \wedge D$. If no contradiction is found, another literal is decided. Otherwise, an asserting clause α is identified. A number of decisions are then erased until we reach the decision level corresponding to the *assertion level* of clause α, at which point α is added to Γ.[6] The solver terminates under one of two conditions: either a contradiction is found under an empty decision sequence D (Line 5), or all literals are successfully decided (Line 12). In the first case, the input CNF must be unsatisfiable. In the second case, the CNF is

[6]The assertion level is computed when the clause is learned. It corresponds to the lowest decision level at which unit resolution is guaranteed to derive a new literal using the learned clause.

Macro : $decide\_literal(\ell, S = (\Delta, \Gamma, D, I))$
$D \leftarrow D; \ell$ `// add a new decision to D`
if *unit resolution detects a contradiction in* $\Delta \wedge \Gamma \wedge D$ **then**
  return *an asserting clause for* (Δ, Γ, D)
$I \leftarrow$ literals implied by unit resolution from $\Delta \wedge \Gamma \wedge D$
return *success*

Macro : $undo\_decide\_literal(\ell, S = (\Delta, \Gamma, D, I))$
erase the last decision ℓ from D
$I \leftarrow$ literals implied by unit resolution from $\Delta \wedge \Gamma \wedge D$

Macro : $at\_assertion\_level(\alpha, S = (\Delta, \Gamma, D, I))$
$m \leftarrow$ assertion level of α
if *there are* m *literals in* D **then return** true
else return false

Macro : $assert\_clause(\alpha, S = (\Delta, \Gamma, D, I))$
$\Gamma \leftarrow \Gamma \cup \{\alpha\}$ `// add learned clause to Γ`
if *unit resolution detects a contradiction in* $\Delta \wedge \Gamma \wedge D$ **then**
  return *an asserting clause for* (Δ, Γ, D)
$I \leftarrow$ literals implied by unit resolution from $\Delta \wedge \Gamma \wedge D$
return *success*

Figure 3: Macros for some SAT-solver primitives.

satisfiable with D as a satisfying assignment.

Algorithm 1 is iterative. Our SDD compiler, however, will be recursive. To further prepare for this recursive algorithm, we will take two extra steps. The first step is to abstract the primitives used in SAT solvers (Figure 3), viewing them as operations on what we shall call a SAT state.

Definition 3. *A SAT state is a tuple $S = (\Delta, \Gamma, D, I)$ where Δ and Γ are sets of clauses, D is a sequence of literals, and I is a set of literals. The number of literals in D is called the decision level of S. Moreover, S is said to be satisfiable iff $\Delta \wedge D$ is satisfiable.*[7]

Here, Δ is the input CNF, Γ is the set of learned clauses, D is the decision sequence, and I are the literals implied by unit resolution from $\Delta \wedge \Gamma \wedge D$. Hence, $\Delta \models \Gamma$ and $D \subseteq I$.

The second step towards presenting our compilation algorithm is a recursive algorithm for counting the models of a CNF, which utilizes the above abstractions (i.e., the SAT state and its associated primitives in Figure 3). To simplify the presentation, we will assume a variable order π of the CNF. If X is the first variable in order π, then one recursively counts the models of $\Delta \wedge X$, recursively counts the models of $\Delta \wedge \neg X$, and then add these results to obtain the model count of Δ. This is given in Algorithm 2, which is called initially with the SAT state $(\Delta, \{\}, \langle\rangle, \{\})$ to count the models of Δ. What makes this algorithm additionally useful for our presentation purposes is that it is *exhaustive* in nature. That is, when considering variable X, it must process both its phases, X and $\neg X$. This is similar to our SDD compilation algorithm — but in contrast to SAT solvers which only consider one phase of the variable. Moreover, Algorithm 2 employs the primitives of Figure 3 in the same way that our SDD compiler will employ them later.

The following is a key observation about Algorithm 2 (and the SDD compilation algorithm). When a recursive call returns a learned clause, instead of a model count, this only means that while counting the models of the CNF $\Delta \wedge D$

[7]Without loss of generality, Δ has no empty or unit clauses.

Algorithm 2: $\#\mathtt{SAT}(\pi, S)$

Input: π : a variable order, S : a SAT state (Δ, Γ, D, I)
Output: Model count of $\Delta \wedge D$ over variables in π, or a clause
1 **if** *there is no variable in* π **then return** 1
2 $X \leftarrow$ first variable in π
3 **if** X *or* $\neg X$ *belongs to* I **then return** $\#\mathtt{SAT}(\pi \backslash \{X\}, S)$
4 $h \leftarrow decide\_literal(X, S)$
5 **if** h *is success* **then** $h \leftarrow \#\mathtt{SAT}(\pi \backslash \{X\}, S)$
6 $undo\_decide\_literal(X, S)$
7 **if** h *is a learned clause* **then**
8   **if** $at\_assertion\_level(h, S)$ **then**
9     $h \leftarrow assert\_clause(h, S)$
10   **if** h *is success* **then return** $\#\mathtt{SAT}(\pi, S)$
11   **else return** h
12   **else return** h
13 $l \leftarrow decide\_literal(\neg X, S)$
14 **if** l *is success* **then** $l \leftarrow \#\mathtt{SAT}(\pi \backslash \{X\}, S)$
15 $undo\_decide\_literal(\neg X, S)$
16 **if** l *is a learned clause* **then**
17   **if** $at\_assertion\_level(l, S)$ **then**
18     $l \leftarrow assert\_clause(l, S)$
19   **if** l *is success* **then return** $\#\mathtt{SAT}(\pi, S)$
20   **else return** l
21   **else return** l
22 **return** $h + l$

targeted by the call, unit resolution has discovered an opportunity to learn a clause (and learned one). Hence, we must backtrack to the assertion level, add the clause, and then try again (Lines 10 and 19). In particular, returning a learned clause does not necessarily mean that the CNF targeted by the recursive call is unsatisfiable. The only exception is the root call, for which the return of a learned clause implies an unsatisfiable CNF (and, hence, a zero model count) since the learned clause must be empty in this case.[8]

5 A Top-Down SDD Compiler

We are now ready to present our SDD compilation algorithm, whose overall structure is similar to Algorithm 2, but with a few exceptions. First, the SDD compilation algorithm is driven by a vtree instead of a variable order. Second, it uses the vtree structure to identify disconnected CNF components and compiles these components independently. Third, it employs a component caching scheme to avoid compiling the same component multiple times.

This is given in Algorithm 3, which is called initially with the SAT state $S = (\Delta, \{\}, \langle\rangle, \{\})$ and a decision vtree v for Δ, to compile an SDD for CNF Δ.[9] When the algorithm is applied to a Shannon vtree node, its behavior is similar to Algorithm 2 (Lines 15–44). That is, it basically uses the Shannon variable X and considers its two phases, X and $\neg X$. However, when applied to a decomposition vtree node v (Lines 5–14), one is guaranteed that the CNF associated with v is decomposed into two components, one associated with

[8]When the decision sequence D is empty, and unit resolution detects a contradiction in $\Delta \wedge \Gamma$, the only learned clause is the empty clause, which implies that Δ is unsatisfiable (since $\Delta \models \Gamma$).

[9]Algorithm 3 assumes that certain negations are freely available (e.g., $\neg p$ on Line 14). One can easily modify the algorithm so it returns both an SDD and its negation, making all such negations freely available. We skip this refinement here for clarity of exposition, but it can be found in the longer version of the paper.

Algorithm 3: $\texttt{c2s}(v, S)$

$unique(\alpha)$ removes an element from α if its prime is $\bot$. It then returns s if $\alpha = \{(p_1, s), (p_2, s)\}$ or $\alpha = \{(\top, s)\}$; returns p_1 if $\alpha = \{(p_1, \top), (p_2, \bot)\}$; else returns the unique SDD node with elements α.

Input: v : a vtree node, S : a SAT state (Δ, Γ, D, I)
Output: A Decision-SDD or a clause

1 **if** *v is a leaf node* **then**
2 　$X \leftarrow$ variable of v
3 　**if** *X or $\neg X$ belongs to I* **then return** *the literal of X that belongs to I*
4 　**else return** $\top$
5 **else if** *v is a decomposition vtree node* **then**
6 　$p \leftarrow \texttt{c2s}(v^l, S)$
7 　**if** *p is a learned clause* **then**
8 　　$clean\_cache(v^l)$
9 　　**return** p
10 　$s \leftarrow \texttt{c2s}(v^r, S)$
11 　**if** *s is a learned clause* **then**
12 　　$clean\_cache(v)$
13 　　**return** s
14 　**return** $unique(\{(p, s), (\neg p, \bot)\})$
15 **else**
16 　$key \leftarrow Key(v, S)$
17 　**if** $cache(key) \neq nil$ **then return** $cache(key)$
18 　$X \leftarrow$ Shannon variable of v
19 　**if** *either X or $\neg X$ belongs to I* **then**
20 　　$p \leftarrow$ the literal of X that belongs to I
21 　　$s \leftarrow \texttt{c2s}(v^r, S)$
22 　　**if** *s is a learned clause* **then return** s
23 　　**return** $unique(\{(p, s), (\neg p, \bot)\})$
24 　$s_1 \leftarrow decide\_literal(X, S)$
25 　**if** *s_1 is success* **then** $s_1 \leftarrow \texttt{c2s}(v^r, S)$
26 　$undo\_decide\_literal(X, S)$
27 　**if** *s_1 is a learned clause* **then**
28 　　**if** $at\_assertion\_level(s_1, S)$ **then**
29 　　　$s_1 \leftarrow assert\_clause(s_1, S)$
30 　　　**if** *s_1 is success* **then return** $\texttt{c2s}(v, S)$
31 　　　**else return** s_1
32 　　**else return** s_1
33 　$s_2 \leftarrow decide\_literal(\neg X, S)$
34 　**if** *s_2 is success* **then** $s_2 \leftarrow \texttt{c2s}(v^r, S)$
35 　$undo\_decide\_literal(\neg X, S)$
36 　**if** *s_2 is a learned clause* **then**
37 　　**if** $at\_assertion\_level(s_2, S)$ **then**
38 　　　$s_2 \leftarrow assert\_clause(s_2, S)$
39 　　　**if** *s_2 is success* **then return** $\texttt{c2s}(v, S)$
40 　　　**else return** s_2
41 　　**else return** s_2
42 　$\alpha \leftarrow unique(\{(X, s_1), (\neg X, s_2)\})$
43 　$cache(key) \leftarrow \alpha$
44 　**return** α

the left child v^l and another with the right child v^r (since v is a decision vtree for Δ). In this case, the algorithm compiles each component independently and combines the results.

We will next show the soundness of the algorithm, which requires some additional definitions. Let Δ be the input CNF. Each vtree node v is then associated with:

– $CNF(v)$: The clauses of Δ mentioning only variables inside the vtree rooted at v (*clauses of* v).

– $ContextC(v)$: The clauses of Δ mentioning some variables inside v and some outside v (*context clauses* of v).

– $ContextV(v)$: The Shannon variables of all vtree nodes that are ancestors of v (*context variables of* v).

– $ContextL(v, I)$: The literals of $ContextV(v)$ appearing in a given set of literals I.

We start with the following invariant of Algorithm 3.

Theorem 2. *Consider a call $\texttt{c2s}(v, S)$ with $S = (., ., D, I)$. Then, $D \subseteq ContextL(v, I)$ and $ContextL(v, I)$ contains exactly one literal for each variable of $ContextV(v)$.*[10]

Hence, when calling vtree node v, all its context variables must be either decided or implied. We can now define the CNF component associated with a vtree node v at state S.

Definition 4. *The component of vtree node v and state $S = (., ., ., I)$ is $CNF(v, S) = CNF(v) \wedge ContextC(v)|\gamma$, where γ are the literals of $ContextV(v)$ appearing in I.*

Hence, the component $CNF(v, S)$ will only mention variables in vtree v. Moreover, the root component ($CNF(v, S)$ with v being the root vtree node) is equal to Δ.

Following is the soundness result assuming no component caching (i.e., while omitting Lines 8, 12, 16, 17 and 43).

Theorem 3. *A call $\texttt{c2s}(v, S)$ with a satisfiable state S will return either an SDD for component $CNF(v, S)$ or a learned clause. Moreover, if v is the root vtree node, then a learned clause will not be returned.*

Theorem 4. *A call $\texttt{c2s}(v, S)$ with an unsatisfiable state S will return a learned clause, or one of its ancestral calls $\texttt{c2s}(v', S')$ will return a learned clause, where v' is a decomposition vtree node.*

We now have our soundness result (without caching).

Corollary 1. *If v is the root vtree node, then call $\texttt{c2s}(v, (\Delta, \{\}, \langle\rangle, \{\}))$ returns an SDD for Δ if Δ is satisfiable, and returns an empty clause if Δ is unsatisfiable.*

We are now ready to discuss the soundness of our caching scheme (Lines 8, 12, 16, 17 and 43). This requires an explanation of the difference in behavior between satisfiable and unsatisfiable states (based on Theorem 1 of Sang *et al.* [2004]). Consider the component CNFs $\Delta_{\mathbf{X}}$ and $\Delta_{\mathbf{Y}}$ over disjoint variables $\mathbf{X}$ and $\mathbf{Y}$, and let Γ be another CNF such that $\Delta_{\mathbf{X}} \wedge \Delta_{\mathbf{Y}} \models \Gamma$ (think of Γ as some learned clauses). Suppose that $I_{\mathbf{X}}$ is the set of literals over variables $\mathbf{X}$ implied by unit resolution from $\Delta_{\mathbf{X}} \wedge \Gamma$. One would expect that $\Delta_{\mathbf{X}} \equiv \Delta_{\mathbf{X}} \wedge I_{\mathbf{X}}$ (and similarly for $\Delta_{\mathbf{Y}}$). In this case, one would prefer to compile $\Delta_{\mathbf{X}} \wedge I_{\mathbf{X}}$ instead of $\Delta_{\mathbf{X}}$ as the former can make unit resolution more complete, leading to a more efficient compilation. In fact, this is exactly what Algorithm 3 does, as it includes the learned clauses Γ in unit resolution when compiling a component. However, $\Delta_{\mathbf{X}} \equiv \Delta_{\mathbf{X}} \wedge I_{\mathbf{X}}$ is not guaranteed to hold unless $\Delta_{\mathbf{X}} \wedge \Delta_{\mathbf{Y}}$ is satisfiable. When this is not the case, compiling $\Delta_{\mathbf{X}} \wedge I_{\mathbf{X}}$ will yield an SDD that implies $\Delta_{\mathbf{X}}$ but is not necessarily equivalent to it. However, this is not problematic for our algorithm, for the following reason. If $\Delta_{\mathbf{X}} \wedge \Delta_{\mathbf{Y}}$ is unsatisfiable, then either $\Delta_{\mathbf{X}}$ or $\Delta_{\mathbf{Y}}$ is unsatisfiable and, hence, either $\Delta_{\mathbf{X}} \wedge I_{\mathbf{X}}$ or $\Delta_{\mathbf{Y}} \wedge I_{\mathbf{Y}}$ will be unsatisfiable, and their conjunction will be unsatisfiable. Hence, even though one of the components was compiled incorrectly, the conjunction remains a valid result. Without component caching, the incorrect compilation will be discarded. However, with component caching, one also

[10] This statement is slightly different than those in [Oztok and Darwiche, 2015]. Yet, it essentialy leads to the same conclusion.

needs to ensure that incorrect compilations are not cached (as observed by Sang *et al.* [2004]).

By Theorem 4, if we reach Line 8 or 12, then state S may be unsatisfiable and we can no longer trust the results cached below v. Hence, $clean\_cache(v)$ on Line 8 and 12 removes all cache entries that are indexed by $Key(v', .)$, where v' is a descendant of v. We now discuss Lines 16, 17 and 43.

Definition 5. *Let v be a vtree node, and let $S = (., ., ., I)$ and $S' = (., ., ., I')$ corresponding SAT states. Let $\mathbf{x} \subseteq I$ (resp., $\mathbf{x}' \subseteq I'$) be the instantiation of variables appearing in both v and $ContextC(v)$. A function $Key(v, S)$ is called a component key iff $Key(v, S) = Key(v, S')$ implies that $CNF(v, S) \wedge \mathbf{x} \equiv CNF(v, S') \wedge \mathbf{x}'$.*[11]

Hence, as long as Line 16 uses a component key according to this definition, then caching is sound. The following theorem describes the component key we used in our algorithm.

Theorem 5. *Consider a vtree node v and a corresponding SAT state $S = (., ., ., I)$. Define $Key(v, S)$ as the following bit vector: (1) each clause δ in $ContextC(v)$ is mapped into one bit that captures whether $I \models \delta$, and (2) each variable X that appears in both vtree v and $ContextC(v)$ is mapped into two bits that capture whether $X \in I$, $\neg X \in I$, or neither. Then function $Key(v, S)$ is a component key.*

6 Experimental Results

We now present an empirical evaluation of the new top-down compiler. In our experiments, we used two sets of benchmarks. First, we used some CNFs from the `iscas85`, `iscas89`, and `LGSynth89` suites, which correspond to sequential and combinatorial circuits used in the CAD community. We also used some CNFs available at *http://www.cril.univ-artois.fr/PMC/pmc.html,* which correspond to different applications such as planning and product configuration. We compiled those CNFs into SDDs and Decision-SDDs. To compile SDDs, we used the SDD package [Choi and Darwiche, 2013a]. All experiments were performed on a 2.6GHz Intel Xeon E5-2670 CPU under 1 hour of time limit and with access to 50GB RAM. We next explain our results shown in Table 1.

The first experiment compares the top-down compiler against the bottom-up SDD compiler. Here, we first generate a decision vtree[12] for the input CNF, and then compile the CNF into an SDD using (1) the bottom-up compiler without dynamic minimization (denoted BU), (2) the bottom-up compiler with dynamic minimization (denoted BU+), and (3) the top-down compiler (denoted TD), using the same vtree.[13] Note that BU+ uses a minimization method, which dynamically searches for better vtrees during the compilation process, leading to general SDDs, whereas both BU and TD do not modify the input decision vtree, hence generating Decision-SDDs with the same sizes. We report the corresponding compilation times and sizes in Columns 2–4 and 6–7, respectively. The top-down Decision-SDD compiler was consistently faster than the bottom-up SDD compiler, regardless of the use of dynamic minimization. In fact, in Column 5 we report the speed-ups obtained by using the top-down compiler against the best result of the bottom-up compiler (i.e., either BU or BU+, whichever was faster). There are 40 cases (out of 61) where we observe at least an order-of-magnitude improvement in time. Also, there are 15 cases where top-down compilation succeeded and both bottom-up compilations failed. However, the situation is different for the sizes, when the bottom-up SDD compiler employs dynamic minimization. In almost all of those cases, BU+ constructed smaller representations. As reported in Column 8, which shows the relative sizes of SDDs generated by TD and BU+, there are 21 cases where BU+ produced an order-of-magnitude smaller SDDs. This is not a surprising result though, given that BU+ produces general SDDs and our top-down compiler produces Decision-SDDs, and that SDDs are a strict superset of Decision-SDDs.

Since Decision-SDDs are a subset of SDDs, any minimization algorithm designed for SDDs can also be applied to Decision-SDDs. In this case, however, the results may not be necessarily Decision-SDDs, but general SDDs. In our second experiment, we applied the minimization method provided by the bottom-up SDD compiler to our compiled Decision-SDDs (as a post-processing step). We then added the top-down compilation times to the post-processing minimization times and reported those in Column 9, with the resulting SDD sizes in Column 10. As is clear, the post-processing minimization step significantly reduces the sizes of SDDs generated by our top-down compiler. In fact, the sizes are almost equal to the sizes generated by BU+ (Column 7). The top-down compiler gets slower due to the cost of the post-processing minimization step, but its total time still dominates the bottom-up compiler. Indeed, it can still be an order-of-magnitude faster than the bottom-up compiler (18 cases). This shows that one can also use Decision-SDDs as a representation that facilitates the compilation of CNFs into general SDDs.

7 Related Work

Our algorithm for compiling CNFs into SDDs is based on a similar algorithm, introduced recently [Oztok and Darwiche, 2014]. The latter algorithm was proposed to improve a size upper bound on SDDs. However, it did not identify Decision-SDDs, nor did it suggest a practical implementation. The current work makes the previously introduced algorithm practical by adding powerful techniques from the SAT literature and defining a practical caching scheme, resulting in an efficient compiler that advances the state-of-the-art.

Combining clause learning and component caching was already used in the context of knowledge compilation [Darwiche, 2004] and model counting [Sang *et al.*, 2004]. Yet, neither of these works described the corresponding algorithms and their properties at the level of detail and precision that we did here. A key difference between the presented top-down compiler and the one introduced in Darwiche [2004], called `c2d`, is that we compile CNFs into SDDs, while `c2d` compiles CNFs into d-DNNFs. These two languages differ

[11] This definition corrects the one in [Oztok and Darwiche, 2015].

[12] We obtained decision vtrees as in Oztok and Darwiche [2014].

[13] Choi and Darwiche [2013b] used balanced vtrees constructed from the natural variable order, and manual minimization. We chose to use decision vtrees as they performed better than balanced vtrees.

	Without post-processing							With post-processing	
	Compilation time				SDD size			Compilation time	SDD size
CNF	BU	TD	BU+	Speed-up	TD	BU+	Ratio	TD+	TD+
c1355	3423.95	189.0	1292.87	6.84	71,642,606	2,430,882	**0.03**	—	—
c432	1.59	0.14	5.62	**11.36**	66,004	13,660	0.21	1.95	14,388
c499	1360.05	31.48	—	**43.20**	29,791,654	—	—	1800.14	3,356,190
c880	3372.87	896.47	—	3.76	214,504,174	—	—	—	—
s1196	763.39	1.86	709.93	**381.68**	2,381,672	245,549	**0.10**	131.53	97,641
s1238	1039.39	2.19	2114.01	**474.61**	1,539,440	139,475	**0.09**	74.42	76,690
s1423	1860.56	5.67	354.62	**62.54**	11,363,370	454,711	**0.04**	588.23	782,464
s1488	564.25	0.57	206.41	**362.12**	457,420	111,671	0.24	19.47	88,671
s1494	2672.46	0.59	1035.91	**1755.78**	465,092	98,812	0.21	21.33	91,690
s510	49.02	0.09	55.38	**544.67**	19,732	10,192	0.52	0.68	7,411
s641	3.84	0.28	4.54	**13.71**	257,322	13,910	**0.05**	5.36	14,623
s713	4.08	0.36	5.91	**11.33**	230,886	13,809	**0.06**	5.22	12,079
s832	80.94	0.33	28.45	**86.21**	501,098	30,841	**0.06**	11.23	28,773
s838	0.71	0.1	4.82	7.10	46,490	9,853	0.21	1.79	13,540
s953	—	1.92	—	**—**	2,772,894	—	—	90.06	161,056
9symml	6.15	0.08	5.29	**66.12**	59,616	15,572	0.26	1.57	14,453
alu2	1164.19	0.13	91.12	**700.92**	114,194	26,866	0.24	2.88	13,093
alu4	—	0.71	—	**—**	2,147,052	—	—	172.81	87,562
apex6	—	235.06	—	**—**	156,430,304	—	—	—	—
frg1	165.61	0.46	22.64	**49.22**	1,551,328	76,632	**0.05**	183.92	123,890
frg2	1876.64	49.76	690.63	**13.88**	21,820,292	235,761	**0.01**	2613.82	1,624,002
term1	517.52	25.36	454.08	**17.91**	5,545,908	249,372	**0.04**	468.92	818,343
ttt2	20.79	0.69	6.54	9.48	468,884	15,328	**0.03**	10.00	18,706
vda	—	0.14	—	**—**	126,152	—	—	11.21	29,266
x4	21.22	0.36	12.04	**33.44**	252,530	23,920	**0.09**	9.16	27,102
2bitcomp_5	16.29	0.35	119.82	**46.54**	337,642	19,289	**0.06**	9.06	58,043
2bitmax_6	—	45.22	—	**—**	153,512,364	—	—	—	—
4blocksb	30.99	168.53	16.85	0.10	1,634	1,989	1.22	168.63	1,530
C163_FW	2457.58	10.55	—	**232.95**	3,909,336	—	—	153.49	84,773
C171_FR	140.77	0.7	92.17	**131.67**	743,212	53,484	**0.07**	69.96	72,415
C210_FVF	1265.00	9.01	—	**140.40**	7,052,986	—	—	426.93	165,582
C211_FS	7.80	0.17	3.93	**23.12**	111,004	8,590	**0.08**	3.00	9,243
C215_FC	—	16.45	—	**—**	11,625,728	—	—	1294.15	431,589
C230_FR	—	32.69	3320.03	**101.56**	38,975,404	571,611	**0.01**	2869.13	763,845
C638_FKA	497.18	5.21	50.35	9.66	1,106,488	17,930	**0.02**	61.95	25,669
ais10	—	2.6	1464.48	**563.26**	61,950	13,940	0.23	4.35	11,997
bw_large.a	62.77	0.01	17.81	**1781.00**	1,512	1,642	1.09	0.16	1,290
bw_large.b	3246.49	0.17	961.77	**5657.47**	5,552	4,309	0.78	0.63	3,698
cnt06.shuffled	2.03	0.04	27.74	**50.75**	3,004	2,874	0.96	0.10	2,994
huge	83.01	0.05	23.79	**475.80**	1,512	1,654	1.09	0.20	1,290
log-1	41.02	0.23	21.39	**93.00**	69,358	6,650	**0.10**	1.99	7,622
log-2	—	8.85	—	**—**	11,249,348	—	—	—	—
log-3	—	4.76	—	**—**	440,868	—	—	185.88	24,418
par16-1-c	224.79	1.22	116.10	**95.16**	1,220	1,204	0.99	1.23	1,214
par16-2-c	356.94	1.26	—	**283.29**	1,362	—	—	1.32	1,242
par16-2	1098.42	1.28	1048.58	**819.20**	3,938	3,938	1.00	1.36	3,922
par16-3	666.46	4.46	713.34	**149.43**	3,960	3,960	1.00	4.54	3,934
par16-5-c	516.75	0.87	—	**593.97**	1,330	—	—	0.93	1,226
par16-5	864.91	4.38	1722.34	**197.47**	3,960	4,000	1.01	4.46	3,934
prob004-log-a	—	181.13	—	**—**	212,553,140	—	—	—	—
qg1-07	—	0.36	—	**—**	4,576	—	—	0.79	2,485
qg2-07	—	0.39	—	**—**	8,072	—	—	1.38	3,992
qg3-08	—	0.15	—	**—**	18,310	—	—	2.69	6,674
qg6-09	—	0.12	—	**—**	6,458	—	—	1.63	4,592
qg7-09	—	0.1	—	**—**	6,712	—	—	1.31	4,004
ra	269.96	4.77	—	**56.60**	619,146	—	—	116.14	342,034
ssa7552-038	4.71	0.14	9.38	**33.64**	44,902	18,786	0.42	1.57	19,147
tire-2	6.98	0.18	5.58	**31.00**	75,472	4,013	**0.05**	1.27	4,487
tire-3	42.13	0.23	26.67	**115.96**	73,914	7,599	**0.10**	1.85	13,038
tire-4	593.53	0.28	98.75	**352.68**	164,996	17,129	**0.10**	5.07	8,395
uf250-026	—	1667.7	—	**—**	8,880	—	—	1667.91	1,013

Table 1: Bottom-up and top-down SDD compilations over `iscas85`, `iscas89`, `LGSynth89`, and some sampled benchmarks. BU refers to bottom-up compilation without dynamic minimization and BU+ with dynamic minimization. TD refers to top-down compilation, and TD+ with a single minimization step applied at the end.

in their succinctness and tractability (SDDs are a strict subset of d-DNNFs, and are less succinct but more tractable). For example, SDDs can be negated in linear time. Hence, the CNF-to-SDD compiler we introduced can easily be used as a DNF-to-SDD compiler. For that, we first negate the DNF into a CNF by flipping the literals and treating each term as a clause. After compiling the resulting CNF into an SDD, we can negate the resulting SDD efficiently, which would become the SDD for the given DNF. Since no efficient negation algorithm is known for d-DNNFs, one cannot use `c2d` when the original knowledge base is represented in DNF. We note that we did not evaluate our compiler for compiling DNFs into SDDs, so we do not know how practical it would be. Still, it can be immediately used to compile DNFs, which has not been discussed before in the context of top-down compilation. Another top-down compiler, called `eadt`, was presented recently [Koriche *et al.*, 2013], which compiles CNFs into a tractable language that makes use of decision trees with `xor` nodes. A detailed comparison of bottom-up and top-down compilation has been made before in the context of compiling CNFs into OBDDs [Huang and Darwiche, 2004]. Our work can be seen as making a similar comparison for compiling CNFs into SDDs.

8 Conclusion

We identified a subset of SDDs, called Decision-SDDs, and introduced a top-down algorithm for compiling CNFs into Decision-SDDs that is based on techniques from the SAT literature. We provided a formal description of the new algorithm with the hope that it would facilitate the development of efficient compilers by the community. Our empirical evaluation showed that the presented top-down compiler can yield significant improvements in compilation time against the state-of-the-art bottom-up SDD compiler, assuming that the input is a CNF.

Acknowledgments

This work has been partially supported by ONR grant #N00014-12-1-0423 and NSF grant #IIS-1118122.

References

[Barrett, 2004] Anthony Barrett. From Hybrid Systems to Universal Plans Via Domain Compilation. In *KR*, pages 654–661, 2004.

[Barrett, 2005] Anthony Barrett. Model Compilation for Real-Time Planning and Diagnosis with Feedback. In *IJCAI*, pages 1195–1200, 2005.

[Beame *et al.*, 2014] Paul Beame, Jerry Li, Sudeepa Roy, and Dan Suciu. Model Counting of Query Expressions: Limitations of Propositional Methods. In *ICDT*, pages 177–188, 2014.

[Bryant, 1986] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[Cadoli and Donini, 1997] Marco Cadoli and Francesco M. Donini. A Survey on Knowledge Compilation. *AI Commun.*, 10(3,4):137–150, 1997.

[Chavira and Darwiche, 2008] Mark Chavira and Adnan Darwiche. On Probabilistic Inference by Weighted Model Counting. *Artif. Intell.*, 172(6-7):772–799, 2008.

[Choi and Darwiche, 2013a] Arthur Choi and Adnan Darwiche. `http://reasoning.cs.ucla.edu/sdd`, 2013.

[Choi and Darwiche, 2013b] Arthur Choi and Adnan Darwiche. Dynamic Minimization of Sentential Decision Diagrams. In *AAAI*, 2013.

[Choi *et al.*, 2013] Arthur Choi, Doga Kisa, and Adnan Darwiche. Compiling Probabilistic Graphical Models using Sentential Decision Diagrams. In *ECSQARU*, pages 121–132, 2013.

[Choi *et al.*, 2015] Arthur Choi, Guy Van den Broeck, and Adnan Darwiche. Tractable Learning for Structured Probability Spaces: A Case Study in Learning Preference Distributions. In *IJCAI*, 2015. To appear.

[Darwiche and Marquis, 2002] Adnan Darwiche and Pierre Marquis. A Knowledge Compilation Map. *JAIR*, 17:229–264, 2002.

[Darwiche, 2004] Adnan Darwiche. New Advances in Compiling CNF into Decomposable Negation Normal Form. In *ECAI*, pages 328–332, 2004.

[Darwiche, 2011] Adnan Darwiche. SDD: A New Canonical Representation of Propositional Knowledge Bases. In *IJCAI*, pages 819–826, 2011.

[Darwiche, 2014] Adnan Darwiche. Tractable Knowledge Representation Formalisms. In Lucas Bordeaux, Youssef Hamadi, and Pushmeet Kohli, editors, *Tractability*, pages 141–172. Cambridge University Press, 2014.

[Elliott and Williams, 2006] Paul Elliott and Brian C. Williams. DNNF-based Belief State Estimation. In *AAAI*, pages 36–41, 2006.

[Herrmann and de Barros, 2013] Ricardo G. Herrmann and Leliane N. de Barros. Algebraic Sentential Decision Diagrams in Symbolic Probabilistic Planning. In *BRACIS*, 2013.

[Huang and Darwiche, 2004] Jinbo Huang and Adnan Darwiche. Using DPLL for Efficient OBDD Construction. In *SAT*, 2004.

[Huang, 2006] Jinbo Huang. Combining Knowledge Compilation and Search for Conformant Probabilistic Planning. In *ICAPS*, pages 253–262, 2006.

[Kisa *et al.*, 2014] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Probabilistic Sentential Decision Diagrams. In *KR*, 2014.

[Koriche *et al.*, 2013] Frédéric Koriche, Jean-Marie Lagniez, Pierre Marquis, and Samuel Thomas. Knowledge Compilation for Model Counting: Affine Decision Trees. In *IJCAI*, 2013.

[Lomuscio and Paquet, 2015] Alessio Lomuscio and Hugo Paquet. Verification of Multi-Agent Systems via SDD-based Model Checking (Extended Abstract). In *AAMAS15*, 2015.

[Marquis, 1995] Pierre Marquis. Knowledge Compilation Using Theory Prime Implicates. In *IJCAI*, pages 837–845, 1995.

[Moskewicz *et al.*, 2001] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, pages 530–535, 2001.

[Oztok and Darwiche, 2014] Umut Oztok and Adnan Darwiche. On Compiling CNF into Decision-DNNF. In *CP*, 2014.

[Oztok and Darwiche, 2015] Umut Oztok and Adnan Darwiche. A Top-Down Compiler for Sentential Decision Diagrams. In *IJCAI*, pages 3141–3148, 2015.

[Palacios *et al.*, 2005] Héctor Palacios, Blai Bonet, Adnan Darwiche, and Hector Geffner. Pruning Conformant Plans by Counting Models on Compiled d-DNNF Representations. In *ICAPS*, pages 141–150, 2005.

[Pipatsrisawat and Darwiche, 2007] Knot Pipatsrisawat and Adnan Darwiche. RSat 2.0: SAT Solver Description. Technical Report D–153, UCLA, 2007.

[Razgon, 2014a] Igor Razgon. Personal communication, 2014.

[Razgon, 2014b] Igor Razgon. On OBDDs for CNFs of Bounded Treewidth. *ArXiv e-prints*, 2014.

[Renkens *et al.*, 2014] Joris Renkens, Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Explanation-Based Approximate Weighted Model Counting for Probabilistic Logics. In *AAAI*, pages 2490–2496, 2014.

[Sang *et al.*, 2004] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining Component Caching and Clause Learning for Effective Model Counting. In *SAT*, 2004.

[Selman and Kautz, 1996] Bart Selman and Henry A. Kautz. Knowledge Compilation and Theory Approximation. *J. ACM*, 43(2):193–224, 1996.

[Vlasselaer *et al.*, 2014] Jonas Vlasselaer, Joris Renkens, Guy Van den Broeck, and Luc De Raedt. Compiling probabilistic logic programs into Sentential Decision Diagrams. In *PLP*, 2014.

[Vlasselaer *et al.*, 2015] Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. Anytime Inference in Probabilistic Logic Programs with Tp-compilation. In *IJCAI*, 2015. To appear.

[Xue *et al.*, 2012] Yexiang Xue, Arthur Choi, and Adnan Darwiche. Basing Decisions on Sentences in Decision Diagrams. In *AAAI*, pages 842–849, 2012.

Quasipolynomial Simulation of Decision-SDDs

In this section we will prove Theorem 1, showing that Decision-SDDs can be simulated by OBDDs with at most a quasipolynomial increase in size. We first start with defining a special type of vtree that will be used in the simulation.

Definition 6 (Right-leaning vtree)**.** *A vtree is called right-leaning when the number of variables in any right subtree is greater than or equal to the number of variables in the sibling left subtree. In a right-leaning vtree, the left child of a decomposition vtree node is called a light node and the right child is called a heavy node. The number of light nodes on a path from the vtree root to one of its leaves is called the light-length of the path. The light-height of the vtree is the maximum light-length attained by any of its paths.*

The following is a key property of right-leaning vtrees that will be used later.

Proposition 2. *Let v be a right-leaning vtree over n variables. The light-height of vtree v is at most $\log n$.*

Proof. Let $v_1, \ldots, v_k$ be the light nodes on the path from the vtree root to a leaf, where v_1 is the closest to the root and v_k is the furthest. Node v_k cannot be a leaf, otherwise its parent would be a Shannon node which is a contradiction with v_k being light. Hence, the vtree rooted at v_k must have ≥ 2 variables. Then, the vtree rooted at v_k's sibling must have ≥ 2 variables since the vtree is right-leaning. Thus, the vtree rooted at v_k's parent must have ≥ 4 variables. Similarly, since v_{k-1} is an ancestor of v_k, the vtree rooted at v_{k-1} must have ≥ 4 variables, and the vtree rooted at its parent must have ≥ 8 variables. Continuing to count this way (by induction), vtree rooted at v_1 must have $\geq 2^k$ variables. Hence, $2^k \leq n$, which implies that $k \leq \log n$. ∎

We next show that one can always efficiently construct a Decision-SDD that respects a right-leaning vtree from one that is not based on a right-leaning vtree. This will ensure that we can always use Decision-SDDs based on right-leaning vtrees instead of arbitrary ones.

Theorem 6. *For every Decision-SDD α of size N, there is another, equivalent Decision-SDD β that respects a right-leaning vtree and has size $p(N)$ for some polynomial p. Moreover, β can be obtained from α in time $O(N)$.*

Proof. Let T be the vtree that α respects. We can obtain β by converting vtree T into a right-leaning one as follows. For each decomposition vtree node v, if the number of variables in the right child is less than those in the left child, we swap the children of vtree node v and adjust the SDD nodes respecting v. As v is a decomposition vtree node, we only need to adjust SDD nodes of the form $\{(f,g),(\neg f,\bot)\}$, $\{(f,\top),(\neg f,g)\}$, and $\{(f,\neg g),(\neg f,g)\}$. The adjustment of these nodes leads to $\{(g,f),(\neg g,\bot)\}$, $\{(g,\top),(\neg g,f)\}$, and $\{(g,\neg f),(\neg g,f)\}$, respectively. This can be done in constant time for each SDD node if the negations of SDD nodes in α are available. This can be ensured by a linear-time preprocessing step that goes over the SDD nodes in α once and negates the nodes along the way. Thus, the whole process to obtain β takes time linear in N and the size of β is $p(N)$ for some polynomial p. ∎

We now prove Theorem 1, which is restated next (with a simple modification that puts right-leaning vtrees into play).

Theorem 1. *Let α be a Decision-SDD of size N that respects a right-leaning vtree v over n variables. Then there exists an equivalent OBDD of size $\leq N^{1+\log n}$ that respects the variable order underlying v.*

Proof. We will appeal to the construction in [Beame *et al.*, 2014], which transforms a DLDD (decomposable logic decision diagrams) into an FBDD, with at most a quasipolynomial increase in size. We first transform SDD α into a DLDD β by transforming the Shannon decision nodes of α into DLDD decision nodes (labeled with Shannon variables), and transforming the decomposition nodes of α into DLDD function nodes (labeled with the functions they represent). Given the correspondence between SDD and DLDD nodes, it is meaningful to talk about the vtree node respected by a DLDD node. It is also meaningful to refer to some DLDD nodes as light or heavy, depending on the vtree node they respect. We will then use the construction of [Beame *et al.*, 2014] to convert β into an FBDD γ, which is then guaranteed to satisfy the following properties:

1. The size of FBDD γ is $\leq N^{1+L} \leq N^{1+\log n}$, where L is the light-height of vtree v.
2. The variable ordering on every path from the root of FBDD γ to one of its leaves is consistent with the left-right variable ordering of vtree v. Hence, FBDD γ is actually an OBDD.

The first property holds because on any path from the root of DLDD β to one of its leaves, the number of light nodes is bounded by the light-height of vtree v, and the light nodes we consider here correspond to the "light edges" used in [Beame *et al.*, 2014]. The second property holds because the construction of [Beame *et al.*, 2014] will end up stacking light nodes over their sibling heavy nodes. In other words, assume there is a DLDD node in β that respects a decomposition vtree node u. Note that its light node l respects some vtree node in u^l and its heavy node h respects some vtree node in u^r. The construction is a recursive one. So, suppose that we already processed nodes l and h, and they are actually OBDDs that respect the variable order underlying u^l and u^r, respectively. The construction will continue by stacking a private copy of l

over h, which will result in an OBDD that is consistent with the left-right variable ordering of u. That is, at each decomposition vtree node u, we will create an OBDD that respects the variable order underlying vtree u. Thus, the resulting structure would respect the variable order underlying vtree v. ■

Soundness of the Compilation Algorithm

We will now present the proofs of the theorems that were used in the soundness of Algorithm 3 (i.e., Theorems 2–4). We start by listing some assumptions/observations that will be used in the rest of the paper. First, given a CNF Δ, we will write $\Delta \vdash I$ to mean that I is the set of literals derived from Δ using unit resolution. Second, S will denote a state (Δ, Γ, D, I), where Δ and Γ are sets of clauses, D is a sequence of literals, and I is a set of literals.[14] Third, each (recursive) call $\texttt{c2s}(v, S)$ of Algorithm 3 will take two inputs v and S such that v is a vtree node belonging to a decision vtree for Δ and S is a SAT state.[15] The latter implies that $\Delta \models \Gamma$ and $\Delta \wedge \Gamma \wedge D \vdash I$. This holds due to the following three facts: (1) initial call is made with $(\Delta, \{\}, \langle\rangle, \{\})$, which is a SAT state as there is no unit or empty clause in Δ; (2) any clause added to Γ is a learned clause, which must be implied by Δ; and (3) literals I is always adjusted before making a call (see SAT primitives in Figure 3). Finally, a SAT state S is said to be *callable* iff unit resolution does not detect a contradiction in $\Delta \wedge \Gamma \wedge D$. Indeed, S will be callable for each call $\texttt{c2s}(v, S)$. This is true due to the following two facts: (1) initial SAT state $(\Delta, \{\}, \langle\rangle, \{\})$ is callable as Δ has no unit or empty clause; and (2) whenever a new SAT state, which is not callable, is constructed during a call, Algorithm 3 backtracks until the contradiction is resolved. So, it will initiate a call only on callable SAT states. We now prove Theorem 2.

Theorem 2. *Consider a call* $\texttt{c2s}(v, S)$ *with* $S = (.,.,D, I)$*. Then,* $D \subseteq ContextL(v, I)$ *and* $ContextL(v, I)$ *contains exactly one literal for each variable of* $ContextV(v)$*.*

Proof. Let $\gamma = ContextL(v, I)$. We first show $D \subseteq \gamma$. For that, we show $D \subseteq I$ and $Vars(D) \subseteq ContextV(v)$. The former is due to $\Delta \wedge \Gamma \wedge D \vdash I$. For the latter, note that when Algorithm 3 decides on a literal (i.e., Line 24 and Line 33), it undoes its decision after completing a recursive call on the next line (i.e., Line 26 and Line 35). Thus, when call $\texttt{c2s}(v, S)$ is made, literals of D must come from recursive calls that are not completed. Indeed, these calls can only be made on the ancestors of v. So, each literal of D must be a literal of some context variable of v. That is, $Vars(D) \subseteq ContextV(v)$. We next show γ contains exactly one literal for each context variable of v. First, we show γ cannot contain two literals of any variable. Assume otherwise. Then, since $\gamma \subseteq I$ and $\Delta \wedge \Gamma \wedge D \vdash I$, unit resolution detects a contradiction in $\Delta \wedge \Gamma \wedge D$, which is a contradiction. We now show γ contains a literal for each context variable X of v. Assume otherwise. Let p be the Shannon node whose Shannon variable is X. Then, variable X is not implied during the corresponding ancestral call to p. As p is a Shannon node, Algorithm 3 will not recurse on p's right child before ensuring X is implied (see Lines 21, 25, and 34). That is, call $\texttt{c2s}(v, S)$ cannot happen, which is a contradiction. So, a literal of X appears in I. Hence, I contains a literal for each context variable of v, and so does γ. ■

We remark that Algorithm 3 is recursive. As such, its execution can be viewed as constructing a tree whose nodes are labeled with recursive calls $\texttt{c2s}(.,.)$, and whose edges are from a recursive call R_1 to another R_2 if R_2 is called within R_1. As some of upcoming proofs will be based on this tree, which we denote by T, we next present two useful observations.

Proposition 3. *Consider an internal node* $\boxed{\texttt{c2s}(v, S)}$ *on* T*. Then,* v *is either a decomposition node or a Shannon node.*

Proof. As $\boxed{\texttt{c2s}(v, S)}$ is an internal node, it must have a child. Then, v cannot be a leaf vtree node, as no recursive call is made on Lines 1–4. So, the proposition follows. ■

Proposition 4. *Consider a leaf node* $\boxed{\texttt{c2s}(v, S)}$ *on* T*. Then, either* v *is a leaf vtree node or* v *is a Shannon node and call* $\texttt{c2s}(v, S)$ *returns a clause on Line 31 or Line 32.*

Proof. Node $\boxed{\texttt{c2s}(v, S)}$ can be a leaf node iff no recursive call happens during call $\texttt{c2s}(v, S)$. Due to Line 6, v cannot be a decomposition node. So, v is either a leaf vtree node or a Shannon node. If v is a Shannon node, then the call can return on one of the following lines: 22, 23, 30, 31, 32, 39, 40, 41, and 44. It is not hard to see that no recursive call happens only when call $\texttt{c2s}(v, S)$ returns a clause on Line 31 or Line 32. ■

To prove Theorem 3 and Theorem 4, we will next present some lemmas.

Lemma 1. *Consider a call* $\texttt{c2s}(v, S)$*. Let* $S' = (\Delta, \Gamma', D', I')$ *be a callable SAT state that appears during call* $\texttt{c2s}(v, S)$*. Then, we have* $ContextL(v, I) = ContextL(v, I')$*.*

Proof. Note that $\Delta \wedge \Gamma \wedge D \vdash I$ and $\Delta \wedge \Gamma' \wedge D' \vdash I'$. We first show $\Gamma \subseteq \Gamma'$ and $D \subseteq D'$, which implies that $I \subseteq I'$. The former holds as Algorithm 3 never erases learned clauses and Γ is obtained before Γ'. The latter holds as call $\texttt{c2s}(v, S)$ does not undo any decision made before its initiation. Hence, $I \subseteq I'$. Let $\gamma = ContextL(v, I)$. Since $\gamma \subseteq I$, $\gamma \subseteq I'$. Also, by Theorem 2, γ contains exactly one literal for each context variable of v. So, I' cannot contain any other literal than γ for context variables of v. Otherwise, unit resolution detects contradiction in $\Delta \wedge \Gamma' \wedge D'$, which violates S' being callable. So, $\gamma = ContextL(v, I')$. ■

Corollary 2. *Consider a call* $\texttt{c2s}(v, S)$*. Let* S' *be a callable SAT state that appears during call* $\texttt{c2s}(v, S)$*. Then, we have* $CNF(v, S) = CNF(v, S')$*.*

Lemma 2. *Let* S *be a SAT state and* v *be a decomposition node of a decision vtree for* Δ*. Then, we have* $CNF(v, S) = CNF(v^l, S) \wedge CNF(v^r, S)$*.*

Proof. Since the input vtree is a decision vtree, there is no clause compatible with v. Thus, we have $CNF(v) = CNF(v^l) \wedge CNF(v^r)$, $ContextC(v) = ContextC(v^l) \wedge$

[14] We will sometimes abuse notation to use D as a set of literals.

[15] All calls considered in the proofs are assumed to be legal (i.e., can be generated by executing Algorithm 3).

$ContextC(v^r)$ and $ContextV(v^l) = ContextV(v^r)$, implying $CNF(v,S) = CNF(v^l,S) \wedge CNF(v^r,S)$. ■

Lemma 3. *Consider a call* $\texttt{c2s}(v,S)$ *on a decomposition node* v. *Let* S' *be a callable state that appears during call* $\texttt{c2s}(v,S)$. *Then,* $CNF(v',S) = CNF(v',S')$ *for* v*'s child* v'.

Proof. $CNF(v,S) = CNF(v,S')$ by Corollary 2. Further, due to Lemma 2, $CNF(v,S) = CNF(v^l,S) \wedge CNF(v^r,S)$ and $CNF(v,S') = CNF(v^l,S') \wedge CNF(v^r,S')$. This implies $CNF(v^l,S) = CNF(v^l,S')$ and $CNF(v^r,S) = CNF(v^r,S')$, and thus the lemma holds. ■

Lemma 4. *Consider a call* $\texttt{c2s}(v,S)$. *Let* $v_1,\ldots,v_n$ *be the decomposition nodes on the path from the vtree root to* v *(excluding* v*) and* v_i' *the child of* v_i *not appearing on the path. Then,* $\Delta|\gamma \equiv \bigwedge_{i=1}^n CNF(v_i',S) \wedge CNF(v,S)$, *where* $\gamma = ContextL(v,I)$.

Proof. Note that $\Delta \equiv \big(\bigwedge_{i=1}^n CNF(v_i') \wedge ContextC(v_i')\big) \wedge CNF(v) \wedge ContextC(v) \wedge \Sigma$, where Σ is a set of clauses that only mention context variables of v. So, the following holds:

$$\begin{aligned}\Delta|\gamma &\equiv \big(\bigwedge_{i=1}^n CNF(v_i')|\gamma \wedge ContextC(v_i')|\gamma\big)\wedge \\ &\qquad CNF(v)|\gamma \wedge ContextC(v)|\gamma \wedge \Sigma|\gamma \\ &\equiv \big(\bigwedge_{i=1}^n CNF(v_i')|\gamma \wedge ContextC(v_i')|\gamma\big)\wedge \\ &\qquad CNF(v)|\gamma \wedge ContextC(v)|\gamma \qquad (1) \\ &\equiv \big(\bigwedge_{i=1}^n CNF(v_i') \wedge ContextC(v_i')|\gamma\big)\wedge \\ &\qquad CNF(v) \wedge ContextC(v)|\gamma \qquad (2) \\ &\equiv \bigwedge_{i=1}^n CNF(v_i',S) \wedge CNF(v,S). \qquad (3)\end{aligned}$$

We now explain why Equations (1)–(3) hold. Equation (1) holds as $\Sigma|\gamma \equiv \top$. To see this, note that γ contains exactly one literal for each context variable of v (see Theorem 2), so that $\Sigma|\gamma \equiv \bot$ or $\Sigma|\gamma \equiv \top$. If $\Sigma|\gamma \equiv \bot$, then unit resolution detects a contradiction in $\Delta \wedge \Gamma \wedge D$ (since $\Delta \wedge \Gamma \wedge D \vdash I$ and $\gamma \subseteq I$). So, $\Sigma|\gamma \equiv \top$. Equation (2) holds as $CNF(v_i')$ and $CNF(v)$ does not mention any context variable of v. Equation (3) holds as $CNF(v_i',S) \equiv CNF(v_i') \wedge ContextC(v_i')|\gamma$ (since $ContextV(v_i') \subseteq ContextV(v)$). ■

Lemma 5. *Consider a call* $\texttt{c2s}(v,S)$. *Then,* $\Delta \wedge D \equiv \Delta \wedge \gamma$ *where* $\gamma = ContextL(v,I)$.

Proof. Note that $\Delta \wedge \Gamma \wedge D \vdash I$. Then, since $\gamma \subseteq I$, $\Delta \wedge \Gamma \wedge D \models \gamma$. Further, since $\Delta \models \Gamma$, $\Delta \wedge \Gamma \wedge D \equiv \Delta \wedge D$. So, $\Delta \wedge D \models \gamma$. By Theorem 2, $D \subseteq \gamma$. So, $\Delta \wedge D \equiv \Delta \wedge \gamma$. ■

Lemma 6. *Consider a call* $\texttt{c2s}(v,S)$. *Then,* S *is satisfiable iff* $\Delta|\gamma$ *is satisfiable where* $\gamma = ContextL(v,I)$.

Proof. By Lemma 5, $\Delta \wedge D \equiv \Delta \wedge \gamma$. Also, as γ is a set of literals, $\Delta \wedge \gamma \equiv \Delta|\gamma \wedge \gamma$, and so $\Delta \wedge D \equiv \Delta|\gamma \wedge \gamma$. Thus, $\Delta \wedge D$ is satisfiable iff $\Delta|\gamma$ is satisfiable. So, the lemma holds. ■

Lemma 7. *Consider a call* $\texttt{c2s}(v,S)$ *on a Shannon node* v *with Shannon variable* X. *Let* $S' = (\Delta,\Gamma',D',I')$ *be a callable SAT state that appears during call* $\texttt{c2s}(v,S)$. *If a literal* ℓ *of* X *appears in* I', *then* $CNF(v^r,S') \equiv CNF(v,S)|\ell$.

Proof. Assume a literal ℓ of X appears in I'. We note that $CNF(v) \wedge ContextC(v) \equiv CNF(v^l) \wedge CNF(v^r) \wedge ContextC(v^r) \wedge \Sigma$, where $\Sigma = ContextC(v^l) \setminus ContextC(v^r)$. Also, $CNF(v^l) \equiv \top$ as v is a Shannon node and there is no unit or empty clause. Then, the following holds, where $\gamma = ContextL(v,I)$:

$$\begin{aligned}CNF(v,S)|\ell &\equiv \big(CNF(v) \wedge ContextC(v)\big)|\gamma\ell \\ &\equiv CNF(v^r) \wedge \big(ContextC(v^r) \wedge \Sigma\big)|\gamma\ell \\ &\equiv CNF(v^r) \wedge ContextC(v^r)|\gamma\ell \qquad (4) \\ &\equiv CNF(v^r,S'). \qquad (5)\end{aligned}$$

We now explain why Equations (4)–(5) hold. Note that $ContextV(v^r) = ContextV(v) \cup \{X\}$ and $\gamma = ContextL(v,I')$ (see Lemma 1). Then, $ContextL(v^r,I') = \gamma\ell$, and so Equation (5) holds. Also, Equation (4) holds since $\Sigma|\gamma\ell \equiv \top$. To see this, note that Σ is defined over context variables of v^r and $\gamma\ell$ contains exactly one literal for each context variable of v^r, which implies that $\Sigma|\gamma\ell \equiv \bot$ or $\Sigma|\gamma\ell \equiv \top$. If $\Sigma|\gamma\ell \equiv \bot$, then unit resolution must detect a contradiction in $\Delta \wedge \Gamma' \wedge D'$ (since $\Delta \wedge \Gamma' \wedge D' \vdash I'$ and $\gamma\ell \subseteq I'$). However, this contradicts with S' being callable. Hence, $\Sigma|\gamma\ell \equiv \top$. ■

Lemma 8. *Consider a call* $\texttt{c2s}(v,S)$ *with a satisfiable state* S. *If a literal* ℓ *of some variable inside* v *appears in* I, *then* $CNF(v,S) \equiv \ell \wedge CNF(v,S)|\ell$.

Proof. Assume a literal ℓ of some variable X inside v appears in I. Note that $\Delta \wedge \Gamma \wedge D \vdash I$. So, $\Delta \wedge \Gamma \wedge D \models \ell$. Further, since $\Delta \models \Gamma$, $\Delta \wedge \Gamma \wedge D \equiv \Delta \wedge D$. So, $\Delta \wedge D \models \ell$. Then, due to Lemma 5, $\Delta \wedge \gamma \models \ell$ where $\gamma = ContextL(v,I)$. Since γ cannot contain ℓ, $\Delta|\gamma \models \ell$. By Lemma 4, $\Delta|\gamma \equiv \Sigma \wedge CNF(v,S)$ where Σ and $CNF(v,S)$ are decomposable CNFs. Here, $CNF(v,S)$ mentions X but Σ does not. Then, given that $\Delta|\gamma$ is satisfiable (see Lemma 6), we have $CNF(v,S) \models \ell$, which implies that $CNF(v,S) \equiv \ell \wedge CNF(v,S)|\ell$. ■

Lemma 9. *Consider a call* $\texttt{c2s}(v,S)$ *with a satisfiable state* S. *Let* ℓ *be a literal of some variable inside* v. *If* $\Delta \wedge D \wedge \ell$ *is unsatisfiable, then* $CNF(v,S)|\ell$ *is unsatisfiable.*

Proof. Assume $\Delta \wedge D \wedge \ell$ is unsatisfiable. Since S is satisfiable, $\Delta \wedge D$ is satisfiable. So, $(\Delta \wedge D)|\ell$ must be unsatisfiable. Then, due to Lemma 5, $(\Delta \wedge \gamma)|\ell$ is unsatisfiable where $\gamma = ContextL(v,I)$. Since γ cannot contain ℓ, $\Delta|\gamma\ell$ is unsatisfiable. Due to Lemma 4, $\Delta|\gamma \equiv \Sigma \wedge CNF(v,S)$ where Σ and $CNF(v,S)$ are decomposable. As $\Delta|\gamma$ is satisfiable (see Lemma 6), Σ is satisfiable. Since Σ does not mention any variable inside v, $\Sigma|\ell \equiv \Sigma$. So, $\Delta|\gamma\ell \equiv \Sigma \wedge CNF(v,S)|\ell$, and hence $CNF(v,S)|\ell$ is unsatisfiable. ■

Lemma 10. *Consider a call* $\texttt{c2s}(v,S)$ *on a leaf node* v *labeled with variable* X. *Then,* $CNF(v,S)$ *is equivalent to one of the following:* $X, \neg X$, *or* $\top$.

Proof. Note that $CNF(v, S) = CNF(v) \wedge ContextC(v)|\gamma$ where $\gamma = ContextL(v, I)$. So, by Theorem 2, $CNF(v, S)$ must be equivalent to one of $X, \neg X, \top$, or $\bot$. We show it cannot be equivalent to $\bot$. Assume otherwise. Note that Δ has neither an empty clause nor a unit clause. So, $CNF(v, S) \equiv ContextC(v)|\gamma$. Thus, $ContextC(v)$ must include two clauses β_1 and β_2 such that $\beta_1|\gamma = X$ and $\beta_2|\gamma = \neg X$. Note that $\Delta \wedge \Gamma \wedge D \vdash I$. Then, since $\gamma \subseteq I$, $\beta_1|\gamma = X$, and $\beta_2|\gamma = \neg X$, unit resolution must detect a contradiction in $\Delta \wedge \Gamma \wedge D$. As this is a contradiction, $CNF(v, S)$ cannot be equivalent to $\bot$. ■

Lemma 11. *Consider a call* $\texttt{c2s}(v, S)$. *If* $CNF(v, S)$ *is unsatisfiable, then call* $\texttt{c2s}(v, S)$ *will return a (learned) clause.*

Proof. Assume $CNF(v, S)$ is unsatisfiable. We use strong induction on the height of node $\boxed{\texttt{c2s}(v, S)}$ on T to show that call $\texttt{c2s}(v, S)$ returns a clause.

Basis: Consider a leaf node $\boxed{\texttt{c2s}(v, S)}$ (i.e., at height 0). Due to Lemma 10, v cannot be a leaf vtree node. Then, by Proposition 4, call $\texttt{c2s}(v, S)$ must return a clause.

Inductive step: As an induction hypothesis (IH), assume that the statement holds for the calls at height less than k where $k \geq 1$. Consider an internal node $\boxed{\texttt{c2s}(v, S)}$ (i.e., at height k). By Proposition 3, v is either a decomposition node or a Shannon node.

Assume v is a decomposition node. Then, $CNF(v, S) = CNF(v^l, S) \wedge CNF(v^r, S)$ due to Lemma 2. Since $CNF(v^l, S)$ and $CNF(v^r, S)$ are decomposable, one of them must be unsatisfiable. Let S^l (resp., S^r) be the SAT state before the call on Line 6 (resp., Line 10). Due to Lemma 3, $CNF(v^l, S^l) = CNF(v^l, S)$ and $CNF(v^r, S^r) = CNF(v^r, S)$. Then, by IH, either Line 6 or Line 10 must construct a clause (whichever component CNF is unsatisfiable), and hence call $\texttt{c2s}(v, S)$ returns a clause on either Line 9 or Line 13.

Assume v is a Shannon node with Shannon variable X. Since $CNF(v, S)$ is unsatisfiable, $CNF(v, S)|\ell$ is unsatisfiable for any literal ℓ. Suppose a literal ℓ of X belongs to I. Then, the call on Line 21 is made with SAT state $S' = S$. By Lemma 7, $CNF(v^r, S') \equiv CNF(v, S)|\ell$, and so is unsatisfiable. Thus, by IH, Line 21 returns a clause, so does call $\texttt{c2s}(v, S)$. Suppose no literal ℓ of X belongs to I. We first show that the condition on Line 27 must be satisfied. This can happen iff Line 24 or Line 25 constructs a clause. Note that if Line 24 does not construct a clause, then the call on Line 25 is made with a SAT state $S' = (., ., ., I')$ where $X \in I'$. By Lemma 7, $CNF(v^r, S') \equiv CNF(v, S)|X$, and so is unsatisfiable. So, by IH, Line 25 returns a clause. That is, the condition on Line 27 must be satisfied. Hence, call $\texttt{c2s}(v, S)$ must return on either Line 30, 31 or 32. As Line 31 and Line 32 both return a clause, it remains to show Line 30 returns a clause. Let S' be the state before the call on Line 30. By Corollary 2, $CNF(v, S) = CNF(v, S')$. So, by IH, Line 30 returns a clause. ■

Lemma 12. *A call* $\texttt{c2s}(v, S)$ *with a satisfiable state* S *will return either an SDD for component* $CNF(v, S)$ *or a (learned) clause.*

Proof. We use strong induction on the height of node $\boxed{\texttt{c2s}(v, S)}$ on T to show that call $\texttt{c2s}(v, S)$ returns either an SDD for $CNF(v, S)$ or a clause.

Basis: Consider a leaf node $\boxed{\texttt{c2s}(v, S)}$ (i.e., at height 0). By Proposition 4, either v is a leaf vtree node or the call returns a clause. Assume v is a leaf vtree node labeled with variable X. Then, by Lemma 10, $CNF(v, S)$ is equivalent to one of $X, \neg X$, or $\top$. So, we simply identify $CNF(v, S)$ on Lines 1–4, and return the corresponding SDD.

Inductive step: As an induction hypothesis (IH), assume that the statement holds for the calls at height less than k where $k \geq 1$. Consider an internal node $\boxed{\texttt{c2s}(v, S)}$ (i.e., at height k). By Proposition 3, v is either a decomposition node or a Shannon node.

Assume v is a decomposition node. So, call $\texttt{c2s}(v, S)$ can return on one of the following lines: 9, 13, or 14. Line 9 and Line 13 return clauses. So, assume the call returns on Line 14. That is, Line 6 and Line 10 do not return clauses. Let S^l (resp., S^r) be the SAT state before the call on Line 6 (resp., Line 10). Since S is satisfiable, both S^l and S^r are satisfiable (note that decision sequence D stays the same). Then, by IH, Line 6 and Line 10 must return SDDs for $CNF(v^l, S^l)$ and $CNF(v^r, S^r)$, respectively. Further, due to Lemma 3, $CNF(v^l, S^l) = CNF(v^l, S)$ and $CNF(v^r, S^r) = CNF(v^r, S)$. Then, since $CNF(v, S) = CNF(v^l, S) \wedge CNF(v^r, S)$ (see Lemma 2), Line 14 returns an SDD for $CNF(v, S)$.

Assume v is a Shannon node with Shannon variable X. So, call $\texttt{c2s}(v, S)$ can return on one of the following lines: 22, 23, 30, 31, 32, 39, 40, 41 or 44. Lines 22, 31, 32, 40, 41 return clauses. So, we study the remaining lines in the following:

[**23**] Here, a literal ℓ of X belongs to I. Then, by Lemma 7, $CNF(v^r, S) \equiv CNF(v, S)|\ell$. So, by IH, Line 21 returns an SDD for $CNF(v, S)|\ell$. Since $CNF(v, S) \equiv \ell \wedge CNF(v, S)|\ell$ (see Lemma 8), Line 23 returns an SDD for $CNF(v, S)$.

[**30, 39**] Let $S' = (., ., D', .)$ be the SAT state before the call on Line 30. It is easy to see that $D' = D$. Then, S' is satisfiable (as S is satisfiable). Also, by Corollary 2, $CNF(v, S') = CNF(v, S)$. So, by IH, Line 30 returns either an SDD for $CNF(v, S)$ or a clause. We can use the same argument for Line 39.

[**44**] To reach this line, calls on Line 25 and Line 34 should not construct clauses. Note that the call on Line 25 should be made with a SAT state $S' = (\Delta, ., DX, .)$. We now show S' is satisfiable. Assume otherwise. That is, $\Delta \wedge D \wedge X$ is unsatisfiable. Then, by Lemma 9, $CNF(v, S)|X$ is unsatisfiable. Note that $CNF(v^r, S') \equiv CNF(v, S)|X$ by Lemma 7. That is, $CNF(v^r, S')$ is unsatisfiable. Then, by Lemma 11, Line 25 returns a clause, which is a contradiction. So, S' is satisfiable. Then, by IH, Line 25 returns an SDD for $CNF(v^r, S')$, which is equivalent to $CNF(v, S)|X$. Similarly, we can show Line 34 returns an SDD for $CNF(v, S)|\neg X$. So, Line 44 returns an SDD for $CNF(v, S)$. ■

Lemma 13. *A call* $\texttt{c2s}(v, S)$ *with a satisfiable state* S *cannot return on neither Line 31 nor Line 40.*

Proof. Assume call $\texttt{c2s}(v, S)$ returns on Line 31. Then, Line 29 must construct a clause. Let $S' = (\Delta, \Gamma', D', .)$ be the SAT state before the call on Line 29. So, unit resolution must detect a contradiction in $\Delta \wedge \Gamma' \wedge D'$. As $\Delta \models \Gamma'$, $\Delta \wedge \Gamma' \wedge D' \equiv \Delta \wedge D'$. So, $\Delta \wedge D'$ is unsatisfiable. Yet, it is easy to see that $D' = D$. That is, $\Delta \wedge D$ is unsatisfiable, which contradicts with S being satisfiable. Thus, call $\texttt{c2s}(v, S)$ cannot return on Line 31. Similarly, we can show that it cannot return on Line 40. ∎

Lemma 14. *Consider a call* $\texttt{c2s}(v, S)$ *with a satisfiable state* S*. If call* $\texttt{c2s}(v, S)$ *returns on either Line 32 or Line 41, then* $D \neq \emptyset$*.*

Proof. Assume call $\texttt{c2s}(v, S)$ returns on Line 32. So, the condition on Line 28 must fail. That is, the current decision level is strictly greater than the assertion level of the learned clause, and hence $D \neq \emptyset$. Similarly, we can show $D \neq \emptyset$ if the call returns on Line 41. ∎

Lemma 15. *Consider a call* $\texttt{c2s}(v, S)$ *with a satisfiable state* S*. If call* $\texttt{c2s}(v, S)$ *returns a (learned) clause, then* $D \neq \emptyset$*.*

Proof. Assume call $\texttt{c2s}(v, S)$ returns a clause. We use strong induction on the height of node $\boxed{\texttt{c2s}(v, S)}$ on T to show that $D \neq \emptyset$.

Basis: Consider a leaf node $\boxed{\texttt{c2s}(v, S)}$ (i.e., at height 0). By Proposition 4, either v is a leaf vtree node or the call returns a clause on Line 31 or Line 32. As call $\texttt{c2s}(v, S)$ returns a clause, v cannot be a leaf node (see Lines 1–4). Also, by Lemma 13, the call cannot return on Line 31. So, by Lemma 14, $D \neq \emptyset$.

Inductive step: As an induction hypothesis (IH), assume that the statement holds for the calls at height less than k where $k \geq 1$. Consider an internal node $\boxed{\texttt{c2s}(v, S)}$ (i.e., at height k). By Proposition 3, v is either a decomposition node or a Shannon node.

Assume v is a decomposition node. As call $\texttt{c2s}(v, S)$ returns a clause, it must return on either Line 9 or Line 13. Suppose it returns on Line 9. So, Line 6 must construct a clause. So, due to IH, $D \neq \emptyset$. Similarly, we can show $D \neq \emptyset$ if the call returns on Line 13.

Assume v is a Shannon node. As call $\texttt{c2s}(v, S)$ returns a clause, it must return on one of the following lines: 22, 30, 31, 32, 39, 40 or 41. By Lemma 13, the call cannot return on Line 31 or Line 40. If the call returns on either Line 32 or Line 41, then $D \neq \emptyset$ by Lemma 14. For the remaining lines (22, 30, 39), using IH, one can easily see that $D \neq \emptyset$. ∎

We are now ready to prove Theorem 3 and Theorem 4.

Theorem 3. *A call* $\texttt{c2s}(v, S)$ *with a satisfiable state* S *will return either an SDD for component* $CNF(v, S)$ *or a learned clause. Moreover, if* v *is the root vtree node, then a learned clause will not be returned.*

Proof. Due to Lemma 12 and Lemma 15 (note that $D = \emptyset$ in the initial SAT state). ∎

Theorem 4. *A call* $\texttt{c2s}(v, S)$ *with an unsatisfiable state* S *will return a learned clause, or one of its ancestral calls* $\texttt{c2s}(v', S')$ *will return a learned clause, where* v' *is a decomposition vtree node.*

Proof. Let $v_1, \ldots, v_n$ be the decomposition vtree nodes on the path from the vtree root to v (excluding v) and v'_i the child of v_i not appearing on the path. By Lemma 4 and Lemma 6, $\bigwedge_{i=1}^{n} CNF(v'_i, S) \wedge CNF(v, S)$ is unsatisfiable. So, one of the (decomposable) components $CNF(v'_i, S)$ and $CNF(v, S)$ is unsatisfiable. Assume $CNF(v, S)$ is unsatisfiable. Then, due to Lemma 11, call $\texttt{c2s}(v, S)$ returns a clause. Assume one of $CNF(v'_i, S)$ is unsatisfiable. Then, by Lemma 2, $CNF(v_i, S)$ is unsatisfiable. Consider the ancestral call $\texttt{c2s}(v_i, S_i)$ of $\texttt{c2s}(v, S)$. By Corollary 2, $CNF(v_i, S_i) = CNF(v_i, S)$, and hence $CNF(v_i, S_i)$ is unsatisfiable. Thus, by Lemma 11, call $\texttt{c2s}(v_i, S_i)$ returns a clause. ∎

Computing Cache Key

We now prove Theorem 5, which is restated next.

Theorem 5. *Consider a vtree node* v *and a corresponding SAT state* $S = (., ., ., I)$*. Define* $Key(v, S)$ *as the following bit vector: (1) each clause* δ *in* $ContextC(v)$ *is mapped into one bit that captures whether* $I \models \delta$*, and (2) each variable* X *that appears in both vtree* v *and* $ContextC(v)$ *is mapped into two bits that capture whether* $X \in I$*,* $\neg X \in I$*, or neither. Then function* $Key(v, S)$ *is a component key.*

Proof. Let $S' = (., ., ., I')$ be a callable SAT state such that $Key(v, S) = Key(v, S')$. We show $CNF(v, S) \wedge \mathbf{x} \equiv CNF(v, S') \wedge \mathbf{x}'$, where $\mathbf{x}$ (resp., $\mathbf{x}'$) is a term of variables appearing in both v and $ContextC(v)$ such that $\mathbf{x} \subseteq I$ (resp., $\mathbf{x}' \subseteq I'$). Due to (2) in the key definition, $\mathbf{x}$ and $\mathbf{x}'$ are the same. So, it is enough to show $CNF(v, S)|\mathbf{x} \equiv CNF(v, S')|\mathbf{x}$. Let $\gamma = ContextL(v, I)$ and $\gamma' = ContextL(v, I')$. By Theorem 2, γ and γ' must include exactly one literal for each variable in $ContextV(v)$. Then, due to (1) in the key definition, we have $ContextC(v)|\gamma\mathbf{x} = ContextC(v)|\gamma'\mathbf{x}$. So, $CNF(v, S)|\mathbf{x} \equiv CNF(v, S')|\mathbf{x}$. ∎