Notes on Validated Model Counting
Version of April 25, 2022

Randal E. Bryant

Computer Science Department
Carnegie Mellon University, Pittsburgh, PA, United States

1 Notation

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of Boolean variables. An *assignment* is a function α assigning Boolean values to the variables: $\alpha : X \rightarrow \{0, 1\}$. We can also view an assignment as a set of *literals* $\{l_1, l_2, \ldots, l_n\}$, where each literal l_i is either x_i or $\overline{x}_i$, corresponding to the assignments $\alpha(x_i)$ = 1 or 0, respectively. We denote the set of all assignments over these variables as $\mathcal{U}$.

1.1 Boolean Functions

A *Boolean function* $f : 2^X \rightarrow \{0, 1\}$ can be characterized by the set of assignments for which the function evaluates to 1: $\mathcal{M}(f) = \{\alpha | f(\alpha) = 1\}$. Let **1** denote the Boolean function that assigns value 1 to every assignment, and **0** denote the assignment that assigns value 0 to every assignment. These are characterized by the universal assignment set $\mathcal{U}$ and the empty assignment set $\emptyset$, respectively.

From this we can define the *negation* of function f as the function $\neg f$ such that $\mathcal{M}(\neg f) = \mathcal{U} - \mathcal{M}(f)$. We can also define the conjunction and disjunction operations over functions f_1 and f_2 as characterized by the sets $\mathcal{M}(f_1 \wedge f_2) = \mathcal{M}(f_1) \cap \mathcal{M}(f_2)$ and $\mathcal{M}(f_1 \vee f_2) = \mathcal{M}(f_1) \cup \mathcal{M}(f_2)$.

For assignment α and a Boolean formula ϕ over X, we use the notation $\alpha[\phi/x_i]$ to denote the assignment α', such that $\alpha'(x_j) = \alpha(x_j)$ for all $j \neq i$ and $\alpha'(x_i) = \alpha(\phi)$, where $\alpha(\phi)$ is the value obtained by evaluating formula E with each variable assigned the value given by α. In particular, the notation $\alpha([\overline{x}_i/x_i])$ indicates the assignment in which the value assigned to x_i is negated, while others remain unchanged.

A Boolean function f is said to be *independent* of variable x_i if every $\alpha \in \mathcal{M}(f)$ has $\alpha([\overline{x}_i/x_i]) \in \mathcal{M}(f)$. The *dependency set* of f, denoted $D(f)$ consists of all variables x_i for which f is *not* independent.

1.2 Separable Cost Functions

Let $\mathcal{Z}$ denote the elements of a commutative ring. A *separable cost function* $\sigma : X \rightarrow \mathcal{Z}$ assigns a value from the ring to each variable. We extend this function by defining the cost of literal $\overline{x}_i$ as $\sigma(\overline{x}_i) = 1 - \sigma(x_i)$, the cost of an assignment as $\sigma(\alpha) = \prod_{l_i \in \alpha} \sigma(l_i)$, and the cost of a function f as $\sigma(f) = \sum_{\alpha \in \mathcal{M}(f)} \sigma(\alpha)$.

For ring elemement a, we use the notation $\sim\!a$ to denote $1 - a$.

Example 1: Let $\mathcal{Z}$ be the set of rational numbers of the form $a \cdot 2^b$ where a and b are integers. Let $\sigma(x_i) = 1/2$ for all variables x_i. The cost of every assignment is then $1/2^n$, and the cost of a function is its *density*, denoted $\rho(f)$. That is, the density of f, for which $0 \leq \rho(f) \leq 1$, is the fraction of assignments for which f evaluates to 1, with $\rho(\mathbf{0}) = 0$ and $\rho(\mathbf{1}) = 1$. The density of a function f can be scaled by 2^n to compute the total number of models $|\mathcal{M}(f)|$. This is the core task of model counting. Using density as the metric, rather than the number of models, has the advantage that it does not vary when the function is embedded in a larger domain $X' \supseteq X$.

Example 2: Let $\mathcal{Z}$ be the set of rational numbers. Assign a *weight* w_i to each variable x_i such that $0 \leq w_i \leq 1$ and let $\sigma(x_i) = w_i$. This implements *weighted model counting* under the restrictions that: 1) the weight of an assignment equals the product of the weights of its literals, and 2) the weight of a variable x_i and its complement $\overline{x}_i$ sum to 1.

Example 3: Let $\mathcal{Z}$ be a field with $|\mathcal{Z}| \geq 2n$, and let $\mathcal{H}$ be the set of functions mapping elements of X to elements of $\mathcal{Z}$. For two distinct functions f_1 and f_2 and a randomly chosen $h \in \mathcal{H}$, the probability that $h(f_1) = h(f_2)$ will be at most $2^n/|\mathcal{Z}| < 1/2$. Therefore, these functions can be used as part of a randomized algorithm for equivalence testing [1].

1.3 Computing Cost Functions

Three key properties of separable cost functions make it possible, in some cases, to compute the cost of a Boolean formula without enumerating all of its satisfying solutions.

Proposition 1 (Complementation). *For separable cost function σ and Boolean function f: $\sigma(\neg f) = 1 - \sigma(f) = \sim\!\sigma(f)$.*

Proposition 2 (Variable-Partitioned Conjunction). *For separable cost function σ and Boolean functions f_1 and f_2 such that $D(f_1) \cap D(f_2) = \emptyset$: $\sigma(f_1 \wedge f_2) = \sigma(f_1) \cdot \sigma(f_2)$.*

We use the notation $f_1 \wedge_{\mathsf{v}} f_2$ to denote the conjunction of f_1 and f_2 under the condition that f_1 and f_2 are defined over disjoint sets of variables.

Proposition 3 (Assignment-Partitioned Disjunction). *For separable cost function σ and Boolean functions f_1 and f_2 such that $\mathcal{M}(f_1) \cap \mathcal{M}(f_2) = \emptyset$: $\sigma(f_1 \vee f_2) = \sigma(f_1) + \sigma(f_2)$.*

We use the notation $f_1 \vee_{\mathsf{a}} f_2$ to denote the disjunction of f_1 and f_2 under the condition that f_1 and f_2 hold for mutually exclusive assignments.

We also find it useful to introduce a restricted version of the if-then-else operator, denoted *ITE*$_{\mathsf{v}}$. For functions f_1, f_2, and f_3, we define *ITE*$_{\mathsf{v}}(f_1, f_2, f_3) = (f_1 \wedge_{\mathsf{v}} f_2) \vee_{\mathsf{a}} (\neg f_1 \wedge_{\mathsf{v}} f_3)$. The two arguments to the disjunction are guaranteed to satisfy the requirements for an assignment-partitioned disjunction, since the first argument can only be satisfied by assignments for which f_1 evaluates to 1, while the second can only be satisfied by assignments for which f_1 evaluates to 0. The only restriction of the *ITE*$_{\mathsf{v}}$ operator is that $D(f_1) \cap D(f_2) = \emptyset$ and $D(f_1) \cap D(f_3) = \emptyset$.

2 Separable Schemas

Table 1. Recursive Definition of Separable Schemas

A) Basic Operations

S	Restrictions	$D(S)$	$\mathcal{M}(S)$
0	None	$\emptyset$	$\emptyset$
1	None	$\emptyset$	$\mathcal{U}$
x_i	None	$\{x_i\}$	$\{\alpha \mid \alpha(x_i) = 1\}$
$\neg S_1$	None	$D(S_1)$	$\mathcal{U} - \mathcal{M}(S_1)$
$S_1 \wedge_{\mathsf{v}} S_2$	$D(S_1) \cap D(S_2) = \emptyset$	$D(S_1) \cup D(S_2)$	$\mathcal{M}(S_1) \cap \mathcal{M}(S_2)$
$S_1 \vee_{\mathsf{a}} S_2$	$\mathcal{M}(S_1) \cap \mathcal{M}(S_2) = \emptyset$	$D(S_1) \cup D(S_2)$	$\mathcal{M}(S_1) \cup \mathcal{M}(S_2)$

B) Composite Operations

S	Definition
$\mathit{ITE}_{\mathsf{v}}(S_1, S_2, S_3)$	$[S_1 \wedge_{\mathsf{v}} S_2] \vee_{\mathsf{a}} [\neg S_1 \wedge_{\mathsf{v}} S_2]$

A *separable schema* is a directed acyclic graph representing a Boolean formula where the only allowed operations are $\neg$, $\wedge_{\mathsf{v}}$, $\vee_{\mathsf{a}}$ and $\mathit{ITE}_{\mathsf{v}}$. We use a graph representation to allow sharing of subformulas. The set of schemas over a set of variables $\{x_1, x_2, \ldots, x_n\}$ can be defined recursively, as is shown in Table 1. Each schema S has an associated dependency set $D(S)$ and an associated set of models $\mathcal{M}(S)$. We divide the operations into *basic* ones that are fundamental to the representation, and *composite* ones that can be constructed from basic operations. The latter category consists of just the $\mathit{ITE}_{\mathsf{v}}$ operation.

A key property of a Boolean formula represented by separable schema S is that, for any separable cost function σ, the cost of the formula $\sigma(S)$ can be computed with a linear number of ring operations.

Table 2 shows a list of *normalizing* transformations to simplify a separable schema. These are mostly straightforward—eliminating extra negations and removing constant terms. The most interesting are the two bottom rules, which use DeMorgan's Laws to simplify an $\mathit{ITE}_{\mathsf{v}}$ operation when one of the arguments is 1 into a $\wedge_{\mathsf{v}}$ operation. We will make use of this transformation when using BDDs to convert a set of clauses into a separable schema.

3 Proof Framework for Cost Functions

The CRAT clausal proof framework provides a means for creating a checkable proof that a Boolean formula, given in conjunctive normal form, is logically equivalent to

Table 2. Normalization Rules

$\neg 0$	$\rightarrow$	1	$\neg 1$	$\rightarrow$	0
$\neg\neg S$	$\rightarrow$	S			
$S \wedge_{\mathsf{v}} 0$	$\rightarrow$	0	$0 \wedge_{\mathsf{v}} S$	$\rightarrow$	0
$S \wedge_{\mathsf{v}} 1$	$\rightarrow$	S	$1 \wedge_{\mathsf{v}} S$	$\rightarrow$	S
$S \vee_{\mathsf{a}} 0$	$\rightarrow$	S	$0 \vee_{\mathsf{a}} S$	$\rightarrow$	S
$S \vee_{\mathsf{a}} 1$	$\rightarrow$	1	$1 \vee_{\mathsf{a}} S$	$\rightarrow$	1
$\mathit{ITE}_{\mathsf{v}}(\neg S_1, S_2, S_3)$	$\rightarrow$	$\mathit{ITE}_{\mathsf{v}}(S_1, S_3, S_2)$			
$\mathit{ITE}_{\mathsf{v}}(1, S_2, S_3)$	$\rightarrow$	S_2	$\mathit{ITE}_{\mathsf{v}}(0, S_2, S_3)$	$\rightarrow$	S_3
$\mathit{ITE}_{\mathsf{v}}(S_1, 1, 0)$	$\rightarrow$	S_1	$\mathit{ITE}_{\mathsf{v}}(S_1, 0, 1)$	$\rightarrow$	$\neg S_1$
$\mathit{ITE}_{\mathsf{v}}(S_1, S_2, 0)$	$\rightarrow$	$S_1 \wedge_{\mathsf{v}} S_2$	$\mathit{ITE}_{\mathsf{v}}(S_1, 0, S_3)$	$\rightarrow$	$\neg S_1 \wedge_{\mathsf{v}} S_3$
$\mathit{ITE}_{\mathsf{v}}(S_1, S_2, 1)$	$\rightarrow$	$\neg[S_1 \wedge_{\mathsf{v}} \neg S_2]$	$\mathit{ITE}_{\mathsf{v}}(S_1, 1, S_3)$	$\rightarrow$	$\neg[\neg S_1 \wedge_{\mathsf{v}} \neg S_3]$

some separable schema. This provides a framework for adding proof generation to model counting programs.

The CRAT format draws its inspiration from the LRAT format for Boolean formulas and the QRAT format for quantified Boolean formulas (QBF). The following are its key properties:

- As with LRAT, a clause can be added as long as either 1) it is blocked, or 2) it satisfies the RAT property with respect to a supplied sequence of earlier antecedent clauses. However, blocked clauses can only be added when defining a schema operation.
- Extension variables can be introduced only according to the operations $\wedge_{\mathsf{v}}$, $\vee_{\mathsf{a}}$, and $\mathit{ITE}_{\mathsf{v}}$.
  - The checker tracks the dependency set for every input and extension variable. When an extension variable is introduced based on the $\wedge_{\mathsf{v}}$ or $\mathit{ITE}_{\mathsf{v}}$ operation, the dependency sets of its arguments must be disjoint.
  - When an extension variable is introduced based on the $\vee_{\mathsf{a}}$ operation, the step must cite earlier steps providing a RAT proof that the two arguments are mutually exclusive.
  - Boolean complement is provided implicitly by allowing the arguments of the extension operations to be literals and not just variables.
- A CRAT proof must show that the schema is logically equivalent to the input formula, not just that they are equisatisfiable. Therefore, each deletion step must also be shown to be equivalence preserving, either because the clause is blocked or it follows from remaining clauses by the RAT property.
- Unlike QRAT, it need not support universal quantification.

3.1 Syntax

Table 3 shows the set of proof rules for the CRAT format. As with other clausal proof formats, a variable is represented by a positive integer v, with the first ones being input variables and successive ones being extension variables. Literal l is represented by a

Table 3. CRAT Step Types. C: clause identifier, L: literal, V: variable

	Rule			Description
C	`i`	L^* `0`		Input clause
C	`ab`	L^+ `0`	$-C^*$ `0`	Add blocked clause
C	`ar`	L^* `0`	C^+ `0`	Add RAT clause
	`db`	C	$-C^+$ `0`	Delete blocked clause
	`dr`	C	C^+ `0`	Delete RAT clause
	`p`	$V\ L\ L$		Declare $\wedge_{\mathsf{v}}$ operation
	`s`	$V\ L\ L$	C^+ `0`	Declare $\vee_{\mathsf{a}}$ operation
	`ite`	$V\ L\ L\ L$		Declare *ITE*$_{\mathsf{v}}$ operation

signed integer, with $-v$ being the complement of variable v. Each clause is indicated by a positive integer identifier C, with the first ones being the IDs of the input clauses and successive ones being the IDs of added clauses. Clause identifiers must be totally ordered, such that clause C can only reference clauses C' such that $C' < C$. Clause identifiers need not be consecutive.

The first set of proof rules are similar to those in other clausal proofs. Our syntax optionally allows input clauses to be listed with a rule of type `i`. Clauses can be added via a blocked-clause addition (command `ab`) or a RAT addition (command `dr`). As described below, however, blocked clauses can only be added to define $\wedge_{\mathsf{v}}$, $\vee_{\mathsf{a}}$, and *ITE*$_{\mathsf{v}}$ operations. The hints portion of a blocked-clause addition lists all earlier clauses containing the negated version of the pivot literal, with the clause IDs negated. The hints portion of a RAT addition must contain a sequence of clause IDs such that the added clause is RAT with respect to these clauses. Similarly, a clause can be deleted if it blocked with respect to one of its literals (command `db`), or because it is RAT (command `dr`) The hints portion of a RAT deletion operation must be an ordered list of clause IDs justifying the deletion.

The second set of proof rules is unique to the CRAT format. Each of these indicates the addition of an extension variable. For each case, the rule must be followed by a sequence of blocked-clause additions providing the defining clauses for the extension variable.

A product rule of the form `p` $v\ l_1\ l_2$ indicates that v will represent the product $l_1 \wedge_{\mathsf{v}} l_2$. The blocked clause additions must encode the formula $v \leftrightarrow (l_1 \wedge l_2)$. Literals l_1 and l_2 must have disjoint dependency sets.

A sum rule of the form `s` $v\ l_1\ l_2$ indicates that v will represent the disjunction $l_1 \vee_{\mathsf{a}} l_2$. The blocked clause additions must encode the formula $v \leftrightarrow (l_1 \vee l_2)$. The rule also contains a sequence of clause IDs such that the clause $\bar{l}_1 \vee \bar{l}_2$ is RAT with respect to the sequence.

An if-then-else rule of the form `ite` $v\ l_1\ l_2\ l_3$ indicates that v will represent *ITE*$_{\mathsf{v}}(l_1, l_2, l_3)$. The blocked clause additions must encode the formula $v \leftrightarrow$ *ITE*$((l_1, l_2, l_3)$. The dependency set for l_1 must be disjoint from those of l_2 and l_3.

3.2 Semantics

A CRAT proof follows the same general form as a QRAT dual proof—one that ensures that each clause addition and each clause deletion preserves equivalence. Starting with the set of input clauses, it produces a sequence of steps that both add and delete clauses. Each addition must be truth preserving and each deletion must be falsehood preserving. At the end, all input clauses must have been deleted, and among the remaining clauses there must be only a single unit clause consisting of some variable or its complement. Except for trivial cases, the final literal will be an extension variable or its complement. That literal will indicate the root of the schema as the (possibly negated) outpuf of a schema operation. Working from that root backward, the schema can be extracted from the CRAT file.

3.3 Example 1

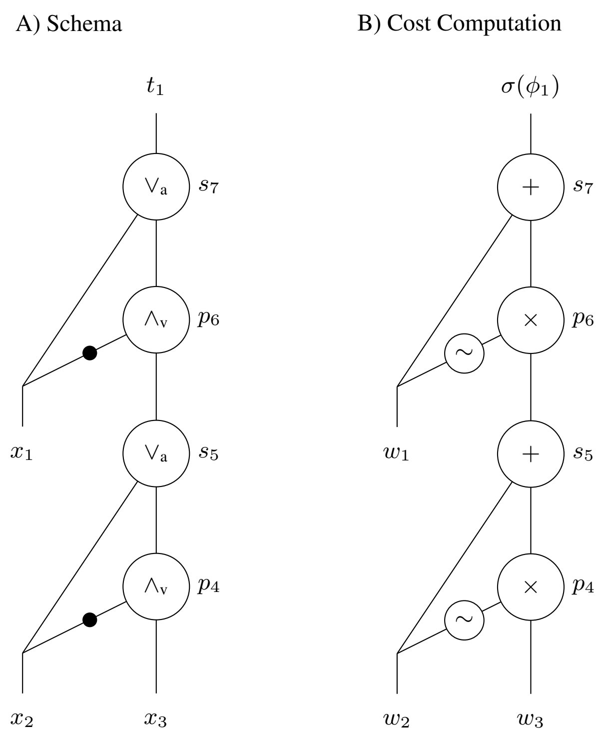

Fig. 1. Schema for Formula $\phi_1 = x_1 \vee x_2 \vee x_3$ and its Cost Computation

As an illustration, consider the Boolean formula $\phi_1 = x_1 \vee x_2 \vee x_3$, represented by a single clause. We cannot directly use the $\vee_{\mathsf{a}}$ operation to form these disjunctions, since the sets of assignments satisfying the individual literals are not disjoint. Instead,

Proof line				Explanation
1	i	1 2 3 0		Input clause
	p	4 -2 3		Declare $p_4 = \overline{x}_2 \wedge_{\mathsf{v}} x_3$
2	ab	4 2 -3 0	0	Defining clauses for p_4
3	ab	-4 -2 0	-2 0	
4	ab	-4 3 0	-2 0	
	s	5 2 4	3 0	Declare $s_5 = x_2 \vee_{\mathsf{a}} p_4$
5	ab	-5 2 4 0	0	Defining clauses for s_5
6	ab	5 -2 0	-5 0	
7	ab	5 -4 0	-5 0	
	p	6 -1 5		Declare $p_6 = \overline{x}_1 \wedge_{\mathsf{v}} s_5$
8	ab	6 1 -5 0	0	Defining clauses for p_6
9	ab	-6 -1 0	-8 0	
10	ab	-6 5 0	-8 0	
	s	7 1 6	9 0	Declare $s_7 = x_1 \vee_{\mathsf{a}} p_6$
11	ab	-7 1 6 0	0	Defining clauses for s_7
12	ab	7 -1 0	-12 0	
13	ab	7 -6 0	-12 0	
14	ar	7 0	12 13 7 6 2 1 0	Assert unit clause $[s_7]$
	dr	1	4 5 10 11 14 0	Delete input clause

Fig. 2. CRAT Proof Steps for Formula $x_1 \vee x_2 \vee x_3$

we must decompose this formula into a sequence of operations. Figure 1A shows one such decomposition. The subscripts of the variables and the operator labels correspond to the numbers of the input and extension variables in the CRAT proof. Edges marked with dots indicate Boolean negation.

The conjunction of $\overline{x}_2$ and x_3 can be computed as $p_4 = \overline{x}_2 \wedge_{\mathsf{v}} x_3$, since the literals have disjoint dependency sets. We can then express the disjunction $x_2 \vee x_3$ as as $s_5 = x_2 \vee_{\mathsf{a}} p_4$. A similar process forms the disjunction $x_1 \vee x_2 \vee x_3$ by first forming the product $p_6 = \overline{x}_1 \wedge_{\mathsf{v}} s_5$ and the final sum $s_7 = x_1 \vee_{\mathsf{a}} p_6$.

The logical representation can readily be converted into a formula for computing $\sigma(\phi_1)$, the cost of formula ϕ_1, given a weight w_i for each variable x_i for $1 \leq i \leq 3$. This is illustrated in Figure 1B. Note how the Boolean negations become $\sim$ operations. This formula is valid for any cost function.

Figure 2 shows an annotated version of the CRAT proof for this example. Clause #1 is the input clause, and clauses #2–#13 are the defining clauses for the four operations. Each of the two sum operations lists one of the earlier defining clauses as a proof that its arguments are mutually exclusive. Proof clause #14 adds the unit clause for the extension variable s_7. We refer to the literal representing formula ϕ_1 as t_1, and we therefore have $t_1 = s_7$. The unit clause indicates that extension variable s_7 will evaluate to 1 for any assigment that satisfies the formula. We can write this as $\phi_1 \vdash t_1$. The deletion step at the end turns this around, showing that $t_1 \vdash \phi_1$, and therefore the input clause can be deleted, This completes a proof that t_1 is logically equivalent to the input formula.

3.4 Example 2

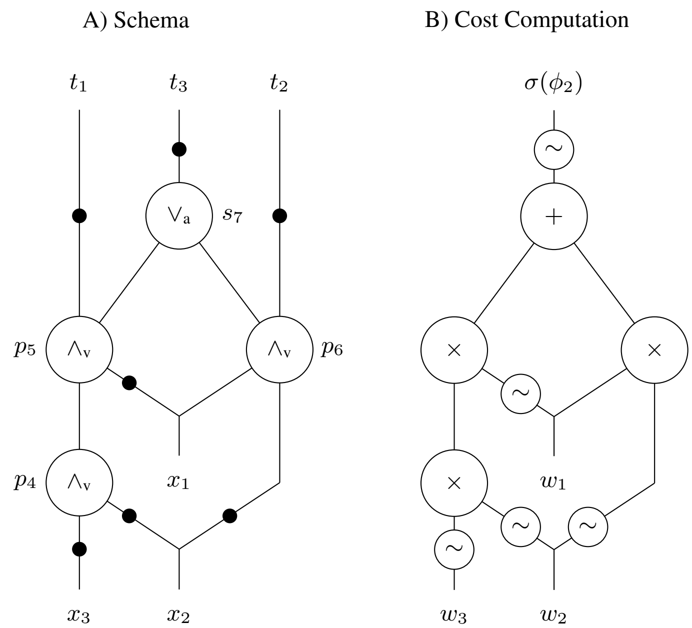

Fig. 3. Schema for Formula $\phi_2 = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2)$ and its Cost Computation

As a more complex example, consider the Boolean formula given by the conjunction of clauses $C_1 = x_1 \vee x_2 \vee x_3$ and $C_2 = \overline{x}_1 \vee x_2$. With this example, we also demonstrate the use of DeMorgan's Laws to provide a more compact encoding of the formula. The complement of the formula can be written in DNF as $\overline{x}_1\,\overline{x}_2\,\overline{x}_3 \vee x_1\,\overline{x}_2$. Each of the two conjuncts can be formed using $\wedge_{\mathsf{v}}$ operations, and their sum can be formed using a $\vee_{\mathsf{a}}$ operation. In this example, we first form a term t_1 that equals C_1 and a term t_2 that equals C_2. These are asserted as unit clauses in proof lines 9 and 13, allowing the input clauses to be deleted. Then we generate t_3 as the conjunction of t_1 and t_2 and assert it as a unit clause. Based on this, we can delete the unit clauses for terms t_1 and t_2. Term t_3 then becomes the root of the schematic representation.

4 Looking Ahead

4.1 Implementing Certified Counters

Given an arbitrary CNF formula, we can use BDD operations to generate a schematic representation. The proof generation can follow the methods we have used for generating unsatisfiability proofs of Boolean formulas [4] and dual proofs of quantified

Proof line				Explanation
1	i	1 2 3 0		Input clause C_1
2	i	-1 2 0		Input clause C_2
	p	4 -2 3		Declare $p_4 = \overline{x}_2 \wedge_{\mathsf{v}} \overline{x}_3$
3	ab	4 2 3 0	0	Defining clauses for p_4
4	ab	-4 -2 0	-2 0	
5	ab	-4 -3 0	-2 0	
	p	5 -1 4		Declare $p_5 = \overline{x}_1 \wedge_{\mathsf{v}} p_4 = \overline{x}_1 \wedge_{\mathsf{v}} \overline{x}_2 \wedge_{\mathsf{v}} \overline{x}_3$
6	ab	5 1 -4 0	0	Defining clauses for p_5
7	ab	-5 -1 0	-6 0	
8	ab	-5 4 0	-6 0	
9	ar	-5 0	7 8 4 5 1 0	Assert $t_1 = \overline{p}_5 = x_1 \vee x_2 \vee x_3$
	dr	1	3 6 7 0	Delete clause C_1
	p	6 1 -2		Declare $p_6 = x_1 \wedge_{\mathsf{v}} \overline{x}_2$
10	ab	6 -1 2 0	0	Defining clauses for p_6
11	ab	-6 1 0	-10 0	
12	ab	-6 -2 0	-10 0	
13	ar	-6 0	11 12 2 0	Assert $t_2 = \overline{p}_6 = \overline{x}_1 \vee x_2$
	dr	2	10 13 0	Delete clause C_2
	s	7 5 6	7 11 0	Declare $s_7 = \overline{t}_1 \vee_{\mathsf{a}} \overline{t}_2$
14	ab	-7 5 6 0	0	Defining clauses for s_7
15	ab	7 -5 0	-5 0	
16	ab	7 -6 0	-5 0	
17	ar	-7 0	9 13 0	Assert $t_3 = \overline{s}_7 = t_1 \wedge t_2$
	dr	9	15 17 0	Delete t_1
	dr	13	16 17 0	Delete t_2

Fig. 4. CRAT Proof Steps for Formula $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2)$

Boolean formulas [3]. The key idea is to use extended resolution to encode the semantics of the BDD nodes as part of the proof. Each BDD node defines an ITE operation. As the transformations of Table 2 indicate, each of these will be represented in the proof by a literal, an $\wedge_{\mathsf{v}}$ operation, or an $\mathit{ITE}_{\mathsf{v}}$ operation. The ITE operation for a node has an input variable x_i as its first argument. In an ordered BDD, the dependency sets of the two children will include only variables greater than x_i in the ordering. Therefore, these operations will satisfy the partitioned-variable requirements of the $\wedge_{\mathsf{v}}$ and $\mathit{ITE}_{\mathsf{v}}$ operations.

Other model counters use either top-down and bottom-up methods to generate representations of the formula that are similar to our schmas. These exploit the key abstractions we have identified. We must find ways to modify these model counters to also generate CRAT proofs.

4.2 TO-DO List

- Proof Framework
  - Generalities and details of the format
  - Can some form of abstraction be incorporated?
    - Want to abstract a subformula to consider only on its cost and dependency set
    - Could represent with fresh extension variable
    - But how to prove logical equivalence?
- Checker
  - Working prototype
  - C/C++ (or Rust?)
  - Formally verified
- Counters
  - BDD-based
    - Prototype
    - C/C++
  - SDD-based
    - Bottom-up
    - Top-down
  - Others?

References

1. Blum, M., Chandra, A.K., Wegman, M.N.: Equivalence of free Boolean graphs can be decided probabilistically in polynomial time. Information Processing Letters **10**(2), 80–82 (18 March 1980)
2. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Computers **35**(8), 677–691 (1986)
3. Bryant, R.E., Heule, M.J.H.: Dual proof generation for quantified Boolean formulas with a BDD-based solver. In: Conference on Automated Deduction (CADE). LNAI, vol. 12699, pp. 433–449 (2021)
4. Bryant, R.E., Heule, M.J.H.: Generating extended resolution proofs with a BDD-based SAT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2021)