Notes on Validated Model Counting
Version of April 11, 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.

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 and empty assignment sets, respectively.

From this we can define the *complement* of function f as the function $\neg f$ such that $\mathcal{M}(\neg f) = \{\alpha | f(\alpha) = 0\}$. 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 E over X, we use the notation $\alpha[E/x_i]$ to denote the assignment α', such that $\alpha'(x_j) = \alpha(x_j)$ for all $j \neq i$ and $\alpha'(x_i) = \alpha(E)$, where $\alpha(E)$ 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 complemented, 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 $\sigma(f) = \sum_{\alpha \in \mathcal{M}(f)} \sigma(\alpha)$.

Example 1: Let $\mathcal{Z}$ be the set of rational numbers and $\sigma(x_i) = 1/2$ 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 satisfies $0 \leq \rho(f) \leq 1$. It is the fraction of assignments for which the 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 $X' \supseteq X$. As a variant, some other forms of weighted model counting can be supported by assigning weights of the form $\sigma(x_i) = w_i$, where $0 \leq w_i \leq 1$ for every variable x_i.

Example 2: Let $\mathcal{Z}$ be a field with $|\mathcal{Z}| > 2n$, 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.

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

Lemma 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 $f_1 \wedge_{\mathrm{pv}} f_2$ 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.

Lemma 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 $f_1 \vee_{\mathrm{pa}} f_2$ to denote the disjunction of f_1 and f_2 under the condition that f_1 and f_2 hold for mutually exclusive assignments.

2 Proof Framework for Cost Functions

We introduce the CRAT clausal proof framework for generating a checkable proof of the valuation of a Boolean formula, given in conjunctive normal form, according to a specified cost function. A CRAT proof provides two forms of information: 1) a *schema* for computing a cost function using the ring operations, and 2) a *proof* that this schema accurately characterizes the cost function for the input formula. A schema can be either *function independent*, meaning that it holds for any cost function, or *function dependent*, meaning that it holds for only for a specific cost function.

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 RUP/RAT property with respect to a supplied sequence of earlier antecedent clauses.
– Extension variables can be introduced only according to the operations $\wedge_{\mathrm{pv}}$, $\vee_{\mathrm{pa}}$, and to an abstraction rule.
  - The checker tracks the dependency set for every input and extension variable. When an extension variable is introduced based on the $\wedge_{\mathrm{pv}}$ operation, the dependency sets of its arguments must be disjoint.
  - When an extension variable is introduced based on the $\vee_{\mathrm{pa}}$ operation, the step must cite earlier steps providing a RUP/RAT proof that the two arguments are mutually exclusive.
  - The *abstraction* operation allows an extension variable to serve as an alias for one of more other variables. Use of this rule renders the schema function dependent.
  - Boolean complement is provided implicitly by allowing the arguments of the extension operations to be literals and not just variables.
– Unlike LRAT, the deletion of a clause requires showing that this clause is implied by one or more other clauses.
– Unlike QRAT, it need not support universal quantification.

2.1 Syntax

Table 1. 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 RUP/RAT clause
	`dr`	C	C^+ `0`	Delete RUP/RAT clause
	`p`	$V\ L\ L$		Declare $\wedge_{\mathrm{pv}}$ operation
	`s`	$V\ L\ L$	C^+ `0`	Declare $\vee_{\mathrm{pa}}$ operation
	`a`	$V\ L$	V^+ `0`	First application of abstraction
	`a`	$V\ L$		Subsequent applications of abstraction

Table 1 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 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 input clauses and successive ones being added clauses. Clause identifiers must be totally ordered, such that clause C can only reference clauses C' such that $C' < C$. However, 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

blocked-clause and RUP/RAT rules. 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 resolution addition must contain a sequence of clause IDs such that the added clause is RUP/RAT with respect to these clauses. Clause deletion requires an explicit justification that the deleted clauses is RUP/RAT with respect to other clauses.

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. A product rule of the form `p` $v\ l_1\ l_2$ indicates that v will represent the product $l_1 \wedge_{\mathrm{pv}} l_2$. 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 $l_1 \vee_{\mathrm{pa}} l_2$. 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 RUP/RAT with respect to the sequence.

An abstraction rule of the form `a` $v\ l$ indicates that variable v will be an alias for literal l. The syntax differs according to whether or not this is the first use of v as an alias. The first instance must include a list of variables that is a (possibly improper) superset of the dependency set for l. It must be followed by a sequence of blocked-clause additions encoding the formula $v \leftrightarrow l$. Subsequent instances do not include an explicit dependency set, but $D(l)$ must be a subset of those declared for v. Furthermore, there are no blocked-clause additions for these cases.

2.2 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 variable will be an extension variable. The sequence of extension operations define the schema for computing the cost function. That is, a value for the cost function is computed for every literal according to the rules of Lemmas 1–3. For every abstraction step, all literals having the indicated extension variable as their alias must evaluate to the same value. The computed cost is then the value computed for the literal in the final unit clause.

2.3 Example

As an illustration, consider the Boolean formula $x_1 \vee x_2 \vee x_3$, represented by a single clause. We cannot directly use the $\vee_{\mathrm{pa}}$ operation to form these disjunctions, since the sets of assignments satisfying the individual literals are not disjoint. Instead, we must decompose this formula into a sequence of operations, as shown by the logical form of Figure 1A. Circles in this schema correspond to $\wedge_{\mathrm{pv}}$ operations and squares to $\vee_{\mathrm{pa}}$

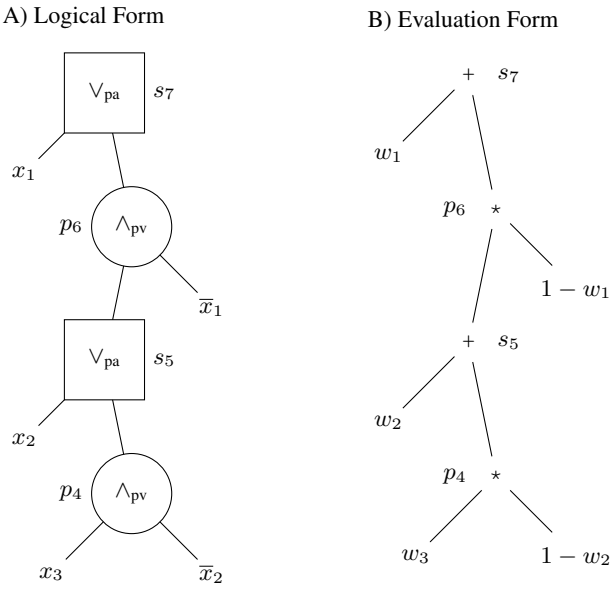

Fig. 1. Schema for Formula $x_1 \vee x_2 \vee x_3$

	Proof line		Explanation
1	i	1 2 3 0	Input clause
	p	4 -2 3	Declare $p_4 = \overline{x}_2 \wedge_{\text{pv}} x_3$
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 $s_5 = x_2 \vee_{\text{pa}} p_4$
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 $p_6 = \overline{x}_1 \wedge_{\text{pv}} s_5$
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 $s_7 = x_1 \vee_{\text{pa}} p_6$
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 $x_1 \vee x_2 \vee x_3$

operations. The subscripts of the variables and the operator labels correspond to the numbers of the input and extension variables in the CRAT proof.

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

The logical representation can readily be converted into a schema for computing the cost of the formula, given weight w_i for each variable x_i for $1 \leq i \leq 3$. This is illustrated in Figure 1B. This schema is valid for any cost function, since it contains no abstraction operations.

Figure 2 shows an annotated version of the CRAT proof for this example. Clause #1 corresponds to the input formula, 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. Clause #14 adds the unit clause corresponding to sum s_7, indicating that the extension variable variable will evaluate to 1 for any assigment that satisfies the input clause. We can write this as $C \vdash s_7$ for input clause C. The deletion step at the end turns this around, showing that $s_7 \vdash C_I$, and therefore the input clause can be deleted, This completes a proof that extension variable s_7 is logically equivalent to the input formula.

3 Looking Ahead

3.1 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 Boolean formulas [3]. The key idea is to use extended resolution to encode the semantics of the BDD nodes as part of the proof. Here we can further decompose each *ITE* (short for "if-then-else") operation representing a BDD node into two $\wedge_{\mathrm{pv}}$ and one $\vee_{\mathrm{pa}}$ operation. That is, we can encode *ITE*(x, A, B) as $(x \wedge_{\mathrm{pv}} A) \vee_{\mathrm{pa}} (\overline{x} \wedge_{\mathrm{pv}} B)$, where x is an input variable and A and B are subformulas. The sum operation trivially satisfies the disjoint assignment requirement, since x has positive polarity in the first product and negative polarity in the second. We must also make sure that x is not in the dependency set of either A or B. For ordered BDDs, [2] this property holds, because the variables in A and B will be greater in the variable ordering than x.

The BDD representations of many of the formulas occuring in model-counting problems are far to large for this approach to be practical. One refinement of the approach, implemented by the ADDMC model counter, is to abstract subformulas, keeping track only of the number of models they can have and representing a number of subformulas with a single ADD leaf node. Our abstraction operation is intended to support this capability. We can introduce a new variable to represent the cost function for some subformula. If subsequent formulas are guaranteed to yield the same cost, then these can alias to the earlier variable. However, it is not clear how to prove that this aliasing preserves equivalence.

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.

3.2 TO-DO List

- Proof Framework
  - Generalities and details of the format
  - What should be the final state?
  - When can blocked clauses be deleted?
  - How can abstraction be incorporated?
- Checker
  - Working prototype
  - C/C++ (or Rust?)
  - Formally verified
- Counters
  - BDD-based
    - Prototype
    - C/C++
  - CDCL-based
    - Prototype
    - C/C++

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)