DRUPing for Interpolants

Arie Gurfinkel
Carnegie Mellon Software Engineering Institute
http://arieg.bitbucket.org

Yakir Vizel
Electrical Engineering Department, Princeton University
Computer Science Department, The Technion
http://www.cs.technion.ac.il/~yvizel

***Abstract*—We present a method for interpolation based on DRUP proofs. Interpolants are widely used in model checking, synthesis and other applications. Most interpolation algorithms rely on a resolution proof produced by a SAT-solver for unsatisfaible formulas. The proof is traversed and translated into an interpolant by replacing resolution steps with AND and OR gates. This process is efficient (once there is a proof) and generates interpolants that are linear in the size of the proof. In this paper, we address three known weakness of this approach: (i) performance degradation experienced by the SAT-solver and the extra memory requirements needed when logging a resolution proof; (ii) the proof generated by the solver is not necessarily the "best" proof for interpolantion, and (iii) combining proof logging with pre-processing is complicated. We show that these issues can be remedied by using DRUP proofs. First, we show how to produce an interpolant from a DRUP proof, even when pre-processing is enabled. Second, we give a novel interpolation algorithm that produces interpolants partially in CNF. Third, we show how DRUP proof can be restructured on-the-fly to yield better interpolants. We implemented our DRUP-based interpolation framework in MiniSAT, and evaluated its affect using AVY — a SAT-based model checking algorithm.**

I. INTRODUCTION

SAT-based Model-Checking, i.e., reducing Model Checking to one or several instances of Boolean satisfiability (SAT), has emerged as the most effective approach for scaling model checking to industrial designs. Bounded Model Checking (BMC) [1] is reduced to a single satisfiabilty problem that checks for existence of a counterexample of a given length. Safety verification (or Unbounded Model Checking) is reduced to an iterative process by repeatedly: (a) solving BMC problems with increasing bound, (b) constructing a proof π of bounded safety, and (c) attempting to generalize π to an inductive invariant. The bounded safety proof π is extracted from the resolution refutation proof of unsatisfiability of a BMC instance by the process of *Craig interpolation*. Thus, safety verification requires that a SAT-solver can produce interpolants in addition to deciding satisfiability.

Formally, given an UNSAT formula $G \equiv A \wedge B$ partitioned into A and B, a Craig interpolant is a formula I such that A implies I, I is inconsistent with B, and I is defined over the variables common to A and B. In model checking, G is a BMC instance, A is some prefix that contains the initial condition, and B a suffix that contains the bad states [2]. Thus, the interpolant I is an over-approximation of the set of states reachable by the prefix A that does not contain any bad states. It is convenient to generalize Craig interpolants to a sequence. In this case, $G \equiv G_1 \wedge \cdots \wedge G_N$ is partitioned into N parts, and an interpolant is a sequence $I_1, \ldots, I_{N-1}$ such that I_i is a Craig interpolant between $G_1 \wedge \cdots \wedge G_i$ and $G_{i+1} \wedge \cdots \wedge G_N$. That is, I_i over-approximates the set of states reachable after i steps. The sequence corresponds to an inductive invariant if for some i, I_i implies $\bigvee_{1 \leq j < i} I_j$. Most SAT-based model checking algorithms (e.g., [2]–[7]) are based in some way on sequence interpolants, although, they vary widely in interpolant computation and in many additional details.

Interpolants can be extracted directly from a resolution proof of unsatisfiability. There are several such *proof-based* procedures that convert a resolution refutation into a circuit by replacing resolution steps by AND and OR gates [2], [8], [9]. They are simple to implement and produce interpolants that are linear in the size of the proof. Their variations (for strength [9], [10], structure [11]–[13], and size [13]) and model checking specific properties (e.g., [9], [14]) are widely studied. However, they require a SAT-solver to log the resolution proof. While this is not technically difficult [15], it significantly increases the memory usage of the solver [16]. Furthermore, it appears that combining proof-logging and common pre-processing is difficult. Most solvers (e.g., [17]) treat proof-logging and pre-processing as mutually exclusive.

Alternatively, interpolants can be constructed by partitioning the clauses of $G \equiv A \wedge B$ into two groups and restricting the SAT-solver to work with either A or B clauses, but not with both at the same time. The solver is allowed to communicate implicants of B to A, and consequences of A to B. The interpolant is the set of all communicated A-consequences. This *proof-less* approach was pioneered by IC3 [5] (together with many other improvements), has been applied for interpolation by Chockler et al. [18], and has been further refined by Bayless et al. [19] by allowing additional communication between partitions. Such algorithms compute interpolants in CNF (which is often desired) and do not need proof-logging. However, partitioning the clauses and restricting the solver significantly degrades performance. This is less of an issue when these techniques are a part of a tightly integrated verification loop, as in IC3. Finally, partitioning negatively affects pre-processing.

Goldberg and Novikov [20] suggest a low-overhead proof logging technique by showing that the sequence of all learnt clauses, in the order learnt by a CDCL SAT-solver – *the clausal proof* – is both easy to log and sufficient to reconstruct the complete resolution proof. Recently, Heule et al. [16] introduced a *trimmed* variant, called *DRUP-proofs*, that additionally account for the clauses deleted by the solver. They show that DRUP-proofs can be expanded (or validated) into a resolution proof efficiently, and suggest them for solver certification, UNSAT core extraction, and interpolation.

In this paper, we propose a novel interpolation algorithm

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0001563.

ISBN: 978-0-9835678-4-4.

based on DRUP-proofs. We are motivated by the fact that logging DRUP-proofs is easy even in the presence of preprocessing. The naïve approach is to expand a DRUP-proof into a resolution proof (e.g., [15]) and apply existing interpolation techniques. While this is reasonable, we take a different, more flexible, approach.

Our contributions are as follows. We present a framework for computing sequence interpolants from DRUP-proofs implemented on top of MiniSAT. The approach consists of two phases. The first traverses the DRUP-proof backward, trimming it, and identifying the core. Unlike [16], our traversal is geared towards interpolation and not proof minimization. The second traverses the trimmed proof forward constructing an interpolant on-the-fly. During this phase, local transformations are applied to the proof to guide it to a better interpolant. It is important to note that our framework is focused on interpolation and not solver certification. Hence, it is made efficient through reuse of many of the solver's data-structures and procedures, and through reuse of the final state of the *trail* when the final conflict is reached. Note that while it seems theoretically trivial to expand a DRUP-proof into a resolution proof, such expansion may take as much time as solving the original SAT instance [16]. Thus, our careful implementation and reuse of the solver's final state is beneficial.

Furthermore, we present a novel interpolation algorithm that computes an interpolant as a pair of formulas $p \wedge g$ such that g is in CNF. In some cases this results in a pure CNF interpolant. To our knowledge, this is unique. Finally, our local proof restructuring, mentioned above, aims at maximizing the CNF component of the interpolant. This restructuring procedure is possible partially due to the flexibility our framework enables when constructing an interpolant.

We evaluated our framework in the context of model checking using AVY [7], a SAT-based model checking algorithm that heavily relies on sequence interpolants. We show the effect our DRUP-based interpolation framework has on AVY's performance when compared to a proof-logging SAT-solver. In addition, we evaluate our different heuristics and show their effect on the computed interpolants. Our experiments show that DRUP-based interpolation is efficient and improves the underlying model checking algorithm. In addition, our new interpolation technique, together with our local proof restructuring result in a significant number of clauses in the CNF component of the computed interpolants.

II. PRELIMINARIES

Given a set U of Boolean variables, a *literal* ℓ is a variable $u \in U$ or its negation $\neg u$, a *clause* is a disjunction of literals, and a formula in *Conjunctive Normal Form*, or a CNF for short, is a conjunction of clauses. It is convenient to treat a clause as a set of literals, and a CNF as a set of clauses. We write $\square$ to denote the empty clause, $Var(\alpha)$ for variables of a clause α, and $Var(G)$ for variables of a set of clauses G.

The *resolution rule* states that given clauses $\alpha_1 = \beta_1 \vee v$ and $\alpha_2 = \beta_2 \vee \neg v$, where β_1 and β_2 are clauses and v and $\neg v$ are literals, one can derive the clause $\alpha_3 = \beta_1 \vee \beta_2$. Application of the resolution rule is denoted by $\alpha_1, \alpha_2 \vdash^{v}_{\text{RES}} \alpha_3$, and v is called the *pivot* variable. We omit v when it is clear from the context or irrelevant.

A *resolution derivation* of a clause α from a CNF formula G is a sequence $\pi = (\alpha_1, \alpha_2, \ldots, \alpha_n \equiv \alpha)$, where each clause α_k is either an *initial* clause of G or is *derived* by applying the resolution rule to clauses α_i, α_j with $i, j < k$. A resolution derivation of the empty clause $\square$ from G is called a *refutation* or a *proof*, and shows that G is unsatisfiable.

A resolution derivation $(\alpha_1, \ldots, \alpha_k)$ is *trivial* [21] if all variables resolved upon are distinct, and each α_i, for $i \geq 3$, is either an initial clause or is derived by resolving α_{i-1} with an initial clause. It is convenient to capture a trivial resolution derivation by a rule. A *chain resolution rule*, written $\alpha_1, \ldots, \alpha_k \vdash^{\vec{x}}_{\text{TVR}} \alpha$, states that α can be derived from $\alpha_1, \ldots, \alpha_k$ by trivial resolution derivation. We call $\alpha_1, \ldots, \alpha_k$ the *chain* and α_1 – the *anchor*, and variables $\vec{x} = (x_1, \ldots, x_{k-1})$ the *chain pivots*. Without loss of generality, we assume that the chain and chain pivots are resolved in the order given. That is, first α_1 is resolved with α_2 on x_1, then the resolvent is resolved with α_3 on x_2, etc. A *chain derivation* is a sequence $\pi \equiv (\alpha_1, \ldots, \alpha_n)$ where each α_k is either an initial clause or is derived by chain resolution from preceding clauses. A *derivation witness* of a chain derivation π is a total function D from clauses of π to sub-sequences of π such that

$$D(\alpha) = [\,] \Rightarrow \alpha \text{ is initial} \quad D(\alpha) \neq [\,] \Rightarrow D(\alpha) \vdash_{\text{TVR}} \alpha \quad (1)$$

Note that a derivation witness is not unique. As usual, a derivation of an empty clause is called a *proof*. Chain proofs capture concisely the proofs produced by CDCL SAT-solvers by logging learned clauses only. For example, the TraceCheck proof format [22] is based on chain derivation.

A *Craig interpolant* [23] of a pair of inconsistent formulas A and B is a formula I such that

$$A \Rightarrow I \quad I \Rightarrow \neg B \quad Var(I) \subseteq Var(A) \cap Var(B) \quad (2)$$

where $Var(A)$ is the set of all variables of A. It is well known that an interpolant can be computed in polynomial time from a resolution proof of unsatisfiability of $A \wedge B$ [2], [8].

For interpolation, it is convenient to partition clauses of a CNF as belonging to A or B. More generally, an *N-colored CNF* is a pair (G, κ) of a CNF formula G and a coloring function $\kappa : G \to [1, .., N]$ that assigns to every clause $\alpha \in G$ a color between 1 and N. We omit the coloring function κ when it is clear from the context or irrelevant and write G for (G, κ). For a colored CNF (G, κ), we write $G_i = \kappa^{-1}(i)$ for the set of all clauses colored i. The coloring extends naturally to variables. For each $v \in Var(G)$, we define its minimum and maximum color as follows:

$$\kappa_{\downarrow}(v) = \min\{i \mid \exists \alpha \in G_i \cdot v \in \alpha\} \quad (3)$$

$$\kappa_{\uparrow}(v) = \max\{i \mid \exists \alpha \in G_i \cdot v \in \alpha\} \quad (4)$$

A variable v is called *local (to partition i)* if $\kappa_{\downarrow}(v) = \kappa_{\uparrow}(v) = i$, and *shared* otherwise. A clause α is *shared* if for all $v \in Var(\alpha)$, v is shared and $\kappa(\alpha) < \kappa_{\uparrow}(v)$. A colored CNF G is *striped* if for all $v \in Var(G)$, $\kappa_{\uparrow}(v) - \kappa_{\downarrow}(v) \leq 1$. That is, every variable is either local, or shared between partitions with adjacent colors. Note that every non-striped CNF can be made striped by adding fresh variables and equality constraints. In the rest of the paper, for simplicity, we assume that all colored CNFs are striped. Given a chain refutation π of a colored

CNF (G, κ) and a derivation witness D of π, we define the maximum color for the clauses of π inductively as follows:

$$\kappa_{\uparrow}(\alpha) = \begin{cases} \kappa(\alpha) & \text{if } \alpha \in G \\ \max\{\kappa_{\uparrow}(\beta) \mid \beta \in D(\alpha)\} & \text{otherwise} \end{cases} \quad (5)$$

Minimum color $\kappa_{\downarrow}(\alpha)$ is defined similarly.

A *sequence (or path) interpolant* for an N-colored unsatisfiable striped CNF (G, κ) is a sequence of formulas $(\top \equiv I_0, \ldots, I_N \equiv \bot)$ such that for all $1 \leq i \leq N$:

$$I_{i-1} \wedge G_i \Rightarrow I_i \quad \forall v \in Var(I_i) \cdot \kappa_{\downarrow}(v) = i \wedge \kappa_{\uparrow}(v) = i + 1$$

We assume that the reader is familiar with the basic CDCL SAT algorithm, as presented in [24]. We assume that the solver maintains all currently implied and decided (i.e., assigned) literals in a queue, called the *trail*, in the order they are assigned. We assume that the solver provides the following API:

- `UnitPropagation` exhaustively applies unit propagation (UP) rule by resolving all unit clauses;
- `ConflictAnalysis` analyzes the most recent conflict and learns a new clause;
- `IsOnTrail` checks whether a clause is in antecedent of a literal on the trail;
- `Enqueue` enqueues one or more literals on the trail;
- `IsDeleted`, `Delete`, `Revive` checks whether a clause is deleted, deletes a clause, and adds a previously deleted clause, respectively;
- `SaveTrail`,`RestoreTrail` save and restore the state of the trail.

III. Trimming Clausal Proofs

Clausal proofs were introduced by Goldberg and Novikov [20] who showed that the sequence of all the learned clauses, in the order they are learned by a CDCL solver, forms a chain derivation. They show that the chain derivation can be validated using UP facilities of the solver. The correctness is based on the following lemma that shows the connection between UP and trivial resolution.

Lemma 1 ([21]) *Given a CNF G and a clause c, c is deducible from G by unit propagation iff c is deducible from G by trivial resolution. That is, $F \vdash_{\text{UP}} c$ iff $F \vdash_{\text{TVR}} c$.*

Two algorithms are suggested in [20], one for backward and one for forward validation. The forward validation replays the proof forward, checking that each clause is subsumed (using UP) by prior clauses. Dually, backward validation walks the proof backwards, removing clauses, and checking that each removed clause is subsumed by the remaining ones.

Recently, backward validation has been improved by Heule et al. [16] who noticed that (a) CDCL solvers aggressively delete unnecessary clauses, and (b) keeping track of clause deletion significantly reduces the number of clauses used by UP during validation. They define a *DRUP-proof* as a sequence $\pi \equiv ((\alpha_0, d_0), \ldots, (\alpha_n, d_n) \equiv (\square, \bot))$, where each d_k is a Boolean flag indicating whether the clause is deleted, and α_k is either an initial clause or is derived by chain resolution from the set of k-active clauses $\{\alpha_j \mid j < k \wedge d_j = \bot \wedge (\forall j < i < k \cdot \alpha_i \neq \alpha_j)\}$. Validation of DRUP-proofs is efficient because validation of a clause α_k depends only on the k-active clauses.

Forward validation walks the proof from the leaves to the empty clause. Thus, it is well suited for interpolation. However, clausal proofs produced by a CDCL solver contain many useless clauses making forward validation inefficient. Heule et al. [16] suggest that in this case, backward validation should be used to trim a clausal proof by removing all clauses that do not contribute to the derivation of the empty clause.

In this section, we present an efficient trimming procedure, called `Trim` and shown in Alg. 1, based on backward validation. Unlike Heule et al., our goal is not to certify a solver, but to trim the proof. Thus, we trust the solver and reuse its intermediate state (namely, the final state of the trail and deletion status of clauses) and routines (namely, unit propagation and conflict analysis). This makes our procedure efficient and easy to implement.

The input to `Trim` is a CDCL solver S in a conflicting state, and a corresponding DRUP-proof π_o. The output is a chain derivation π such that all clauses of π participate in a derivation of the empty clause. In the terminology of Heule et al., all clauses of π are *core*. The algorithm maintains a set C of core clauses. It walks the input DRUP-proof π_o backwards. Deleted clauses are revived (line 3). If the current clause α_i is on the trail, `UndoTrailCore` is used to pop the literals of the trail up to and including the literal whose antecedent is α_i. In the process, antecedents of any core literal on the trail are marked core as well. Next, α_i is removed from the solver, and, if it is not initial, validated using UP. For that, the negation of the literals is put on a trail and `UnitPropagate` is used to derive the conflict. Note that this always succeeds since we assume that the solver S and the proof π_o are valid. Finally, `ConflictAnalysisCore` is used to analyze the conflict, and, in the process, marks all clauses in the implication graph of the conflict as core. When the main loop terminates, π is a chain proof in reversed order.

We use `Trim` to trim a DRUP-proof before interpolation using forward validation. In the rest of the paper, we assume that all chain proofs are trimmed. The interpolation procedure is described in Section IV. `Trim` provides two degrees of freedom. First, different UP strategies result in different proofs. For example, Heule et al. prefer core clauses during UP to minimize the total size of the trimmed proof. Second, `ConflictAnalysisCore` can introduce additional clauses corresponding to different cuts of the implication graph. We propose strategies that result in better interpolants in Section V.

IV. Interpolation Algorithm

In this section, we present our interpolation algorithm.

Let (G, κ) be an N-colored striped CNF formula. Throughout this section, we assume, for simplicity, that $N = 3$. However, our results easily extend to an arbitrary number of colors. We denote shared variables of partition j by $V_j = Var(G_j) \cap Var(G_{j+1})$. For a clause $\alpha \in G$, we write $\alpha|_{[k,l]}$ for a clause obtained from α by removing all variables v with color less than k ($\kappa_{\uparrow}(v) < k$) or greater than l ($\kappa_{\uparrow}(v) > l$). We write $\alpha|_{\leq l}$ for $\alpha|_{[1,l]}$ and $\alpha|_{\geq k}$ for $\alpha|_{[k,N]}$. Recall that a clause α is *shared* w.r.t. j if $Var(\alpha) \subseteq V_j$ and $\kappa(\alpha) = j$.

Algorithm 1: `Trim`(S, π_o)

Input: A SAT-solver instance S with $\square$ on the trail and the corresponding DRUP-proof $\pi_o = ((\alpha_0, d_0), \ldots, (\alpha_n, \bot) \equiv (\square, \bot))$
Output: A chain derivation $(\beta_0, \ldots, \beta_m \equiv \square)$

```
1  π = [ ] ; C = {α_n}
2  for i = n to 0 do
3      if S.IsDeleted(α_i) then S.Revive(α_i)
4      else
5          if S.IsOnTrail(α_i) then
6              S.UndoTrailCore(α_i, C)
7          S.Delete(α_i)
8          if α_i ∈ C then
9              if α_i is not initial then
10                 S.SaveTrail()
11                 S.Enqueue(¬α_i)
12                 c = S.UnitPropagation()
13                 S.ConflictAnalysisCore(c, C)
14                 S.RestoreTrail()
15             π.Append(α_i)
16 Reverse(π)
```

Our procedure, called `ChainItp`, is shown in Alg. 2. The inputs are a N-colored CNF (G, κ) and a (trimmed) chain derivation π. The output is a sequence interpolant $I_0, \ldots, I_N$. `ChainItp` walks π forward from α_0 to α_n and computes partial interpolants for each partition (or color) separately. For partition i and a clause α_j, a partial interpolant is a conjunction of a pair of formulas $p_i(\alpha_j) \wedge g_i$. g_i contains the CNF part of the interpolant, and $p_i(\alpha_j)$ contains the rest. The final interpolant is obtained as a partial interpolant of the empty clause $\alpha_n \equiv \square$.

For a fixed color k, we partition the clauses of π into two groups: *leaf* and *non-leaf*. A clause is a leaf (for color k) if it is either initial, or derived only using clauses with color less than or equal to k. Otherwise, it is non-leaf. The leaf and non-leaf clauses are interpolated using helper functions `Leaf` and `Tvr`, respectively. Before going into detail, let us introduce the following notion:

Definition 1 Let (G, κ) be an N-colored striped CNF formula, π a chain refutation of G, D a derivation witness for π, and k a natural number $1 \leq k \leq N$. A shared leaf $\alpha \in \pi$ is *shared-derivable* w.r.t. k and D if for all $\beta \in D(\alpha)$, $\kappa_{\downarrow}(\beta) = k$ or β is shared-derivable w.r.t. $k-1$ and D.

Clearly, for initial shared clauses, this definition holds trivially. Intuitively, α is shared-derivable w.r.t. k if it is derived using only clauses from G_k and shared-derivable clauses w.r.t. $k-1$. Let us assume that our stripped CNF formula is $G_1 \wedge G_2 \wedge G_3$. All shared clauses w.r.t. G_1 are also shared-derivable. A shared clause w.r.t. G_2 is shared-derivable w.r.t. 2 iff it is derived using clauses from G_2 and clauses that are shared-derivable w.r.t. G_1. Note that we maintain a derivation witness D as part of the definition due to the fact that a chain derivation represent a space of possible resolution steps that may lead to a derived clause. Thus, in order for our recursive definition to apply, we must make sure a specific derivation witness is used.

Lemma 2 *Let (G, κ) be an N-colored striped CNF formula. Given a chain derivation π, let D be a derivation witness of π. Let $(g_0 = \top, g_1, \ldots, g_N)$ be a sequence such that g_i is a CNF containing all shared-drivable clauses w.r.t. a color i and D, then $g_{i-1} \wedge G_i \Rightarrow g_i$ for $1 \leq i \leq N$.*

The proof is immediate from the definition of shared-derivable clauses.

We now go into more detail about the mechanics of `Leaf` and `Tvr`. The function `Leaf` is applied to initial clauses (line 4) and to derived leaf clauses (line 15). The input is a clause α, a color j and a derivation witness D. The output is a pair (p, g) such that $p \wedge g$ is a partial interpolant of α for color j, and g is in CNF. It works according to the following rules:

- if α is shared-derivable w.r.t. j and D: $p = \top$ and $g = \alpha$.
- otherwise, if $\kappa(\alpha) \leq j$ then $p = \alpha|_{\geq j+1}$ and $g = \top$
- otherwise, $p = g = \top$

The function `Tvr` is applied to derived clauses. The input is a clause α, a corresponding chain derivation $\vec{\beta} \vdash^{\vec{x}}_{\text{TVR}} \alpha$, and a color j. The chain $\vec{\beta} = (\beta_0, \ldots, \beta_b)$ is obtained by UP and conflict analysis (lines 8-10) as described in Section III. The output is a formula q_b, where

$$q_l = \begin{cases} p_j(\beta_0) & \text{if } l = 0 \\ q_{l-1} \bowtie^j_{x_l} p_j(\beta_l) & \text{ow} \end{cases} \qquad \bowtie^j_x = \begin{cases} \wedge & \text{if } \kappa_{\uparrow}(x) \leq j \\ \vee & \text{ow} \end{cases}$$

That is, `Tvr` walks up the chain $\vec{\beta}$, and, at each resolution step, either conjoins or disjoins the partial interpolants of the chain clauses. `Tvr` is effectively a direct extension of the interpolation rules of [2] from resolution to chain resolution.

It is important to note that the derivation witness D is not stored explicitly in our implementation of the algorithm, and it is used implicitly by `Leaf`. We only mention it in Algorithm 2 for clarity.

Our interpolation algorithm is somewhat unorthodox since it treats some of the derived clauses as leaves. Furthermore, it keeps a CNF part of the interpolant separately (using g_j). We show that none-the-less, it still produces a valid sequence interpolant.

Definition 2 Given an unsatisfiable N-colored striped CNF (G, κ) and a chain derivation π. A sequence of partial interpolants $(\top, p_1, \ldots p_{N-1}, \bot)$ and a set of CNF formulas $\{g_j\}_1^{N-1}$ are *valid* iff for every $1 \leq k \leq N$, and for every $\alpha \in \pi$, $(p_k(\alpha) \wedge g_k \wedge G_{k+1}) \Rightarrow (p_{k+1}(\alpha) \vee \alpha|_{\geq k+1}) \wedge g_{k+1}$.

Note that a valid partial interpolant sequence results in a valid sequence interpolant. We show that the partial interpolants of `ChainItp` satisfy validity requirement of Def 2.

Theorem 1 *Given an N-colored striped CNF (G, κ) and a chain derivation π, the sequence of partial interpolants $(\top, p_1, \ldots p_{N-1}, \bot)$ and the set of CNF formulas $\{g_j\}_1^{N-1}$ computed by* `ChainItp` *are valid.*

Proof: For simplicity, we show the proof for the case $N = 3$. The proof for the general case is similar. Furthermore, we rely on the fact that without our special leaf handling, `ChainItp` is a straightforward extension of McMillan's procedure [2] to chain resolution. We use $q_j(\alpha)$ to denote the partial interpolant of [2].

Algorithm 2: `ChainItp`

Input: A SAT-solver instance S, colored CNF (G, κ), $\kappa : G \to [1..N]$, and a chain derivation $\pi = (\alpha_0, \ldots, \alpha_n \equiv \bot)$

Output: An interpolation sequence $(\top \equiv I_0, I_1, \ldots, I_N \equiv \bot)$

```
for i = 0 to n do
    if α_i ∈ G then
        for j = 1 to N − 1 do
            (p_j(α_i), g) ← Leaf(α_i, j)
            g_j ← g_j ∧ g
    else
        S.UnitPropagate(), S.SaveTrail()
        S.Enqueue(¬α_i)
        β_0 = S.UnitPropagate()
        β⃗ = S.ConflictAnalysisTvr(β_0, α_i)
            /* β⃗ = (β_0, ..., β_b) is a subsequence
            of π s.t. β⃗ ⊢^{x⃗}_{TVR} α_i */
        D(α_i) ← β⃗
        κ(α_i) ← max{κ(c) | c ∈ β⃗}
        for j = 1 to N − 1 do
            if κ(α_i) ≤ j then
                (p_j(α_i), g) ← Leaf(α_i, j)
                g_j ← g_j ∧ g
            else
                p_j(α_i) ← Tvr(β⃗, α_i, j)
        S.RestoreTrail()
        S.Revive(α_i)
I_0 ← ⊤, I_N ← ⊥
for j = 1 to N − 1 do  I_j ← p_j(α_n) ∧ g_j
```

The proof is by induction on the graph induced by π and D. The base case follows from [14] since for an initial clause α $p_j(\alpha) \wedge g_j = q_j(\alpha)$. For the inductive step, we only consider the case of a single resolution step. Let c_1 and c_2 be two clauses that resolve on v to get c. W.l.o.g., assume $v \in c_1$ and $\neg v \in c_2$. By inductive hypothesis:

$$p_1(c_1) \wedge g_1 \wedge G_2 \Rightarrow (p_2(c_1) \vee c_1|_{\geq 2}) \wedge g_2 \quad (6)$$
$$p_1(c_2) \wedge g_1 \wedge G_2 \Rightarrow (p_2(c_2) \vee c_2|_{\geq 2}) \wedge g_2 \quad (7)$$

Since we rely on [2], [14], we only need to prove the correctness for our modifications, namely treating derived clauses as leaves. Thus, there are only two cases: (1) c is derived using only clauses from G_1, or (2) c is derived using only clauses from G_1 and G_2. Case (1) is immediate by Lemma 2. For case (2), w.l.o.g., assume that c_1, c_2, and c are not leaves w.r.t. 1, but are leaves w.r.t. 2. In this case, we can substitute p_2 with a partial interpolant for the leaf. The induction hypothesis becomes:

$$p_1(c_1) \wedge g_1 \wedge G_2 \Rightarrow (c_1|_{\geq 2}) \wedge g_2 \quad (8)$$
$$p_1(c_2) \wedge g_1 \wedge G_2 \Rightarrow (c_2|_{\geq 2}) \wedge g_2 \quad (9)$$

By the definition of p_1 we know that if $v \in \mathit{Var}(G_3)$ then $p_1(c) = p_1(c_1) \wedge p_1(c_2)$, otherwise $p_1(c) = p_1(c_1) \vee p_1(c_2)$. We take care of the following two cases. Case 1, c is shared-derivable. We need to show that $(p_1(c) \wedge g_1) \wedge G_2 \Rightarrow (\top \vee c) \wedge g_2$. Since c is shared-derivable $c \in g_2$ and g_1 is unchanged. By Lemma 2, $p_1(c) \wedge g_1 \wedge G_2 \Rightarrow g_2$ holds.

Case 2, c is not shared-derivable. We need to show that $(p_1(c) \wedge g_1) \wedge G_2 \Rightarrow (c|_{\geq 3} \vee c|_{\geq 2}) \wedge g_2$. Since c is not shared-derivable both g_1 and g_2 are unchanged. Assume that $v \notin \mathit{Var}(G_3)$, then $p_1(c) = p_1(c_1) \vee p_1(c_2)$. Assume, to the contrary, that $p_1(c) \wedge g_1 \wedge G_2 \Rightarrow c|_{\geq 2} \wedge g_2$ does not hold. Then, there is an assignment s.t. $(p_1(c) \wedge g_1) \wedge G_2$ evaluates to $\top$ while $c|_{\geq 2} \wedge g_2$ evaluates to $\bot$. From Lemma 2, we know that g_2 evaluates to $\top$, therefore, $c|_{\geq 2}$ is $\bot$. W.l.o.g. assume that under this assignment $p_1(c_1)$ evaluates to $\top$. By the induction hypothesis $c_1|_{\geq 2} \wedge g_2$ evaluates to $\top$ as well. Due to our assumption that $c|_{\geq 2}$ evaluates to $\bot$, v must evaluate to $\top$. but, since $v \in c_1|_{\geq 2}$, it must aslo be part of $c|_{\geq 2}$. Thus, indicating that $c|_{\geq 2}$ evaluates to $\top$, in contradiction to our assumption. The other cases are proved similarly. ■

V. Colors, Proofs, and CNF

In this section, we discuss how to combine our framework with a light-weight proof restructuring. The goal of restructuring is to increase the number of shared derived leaves in the proof to increase the CNF component of the interpolant. We first introduce the concept of colorable chain refutations and show that they lead to a simple CNF interpolation procedure. However, an ordinary chain refutation is exponentially stronger than colorable one. Hence, restricting to colorable refutations is not practical. Instead, we propose a polynomial algorithm to restructure a refutation on-the-fly to increase its colaribility.

Let (G, κ) be a striped N-colored CNF, π a chain refutation of G, and D a derivation witness for π. The witness D is called *colored* if for every derived clause $\alpha \in \pi$, the corresponding derivation sequence $D(\alpha) = (\beta_0, \ldots, \beta_n)$ satisfies the following condition: for all $0 \leq i \leq n$, $\kappa_\uparrow(\beta_i) = \kappa_\downarrow(\beta_i) = \kappa_\uparrow(\alpha)$ or $\kappa_\uparrow(\beta_i) < \kappa_\uparrow(\alpha)$ and β_i is shared. A chain refutation π is *colorable* if there exists a colored refutation witness for it.

Colorable refutations induce a simple interpolation procedure. Let $\pi = (\alpha_0, \ldots, \alpha_n)$ be a colorable chain refutation with N colors. Then, the sequence $\vec{I} \equiv (I_0 \equiv \top, \ldots, I_N \equiv \bot)$ defined as follows:

$$I_i = \{\alpha \in \pi \mid \kappa_\uparrow(\alpha) = i \wedge \alpha \text{ is shared}\} \quad (10)$$

is a sequence-interpolant. Furthermore, $\vec{I}$ is in CNF and is linear in the size of the chain refutation π. This is not a coincidence. Colorable chain refutations and CNF interpolants are closely related.

Theorem 2 *For every colorable chain refutation π of N-colored CNF G there exists a sequence interpolant $\vec{I}$ such that $\sum_{i=0}^{N} |I_i| < |\pi|$. For every CNF sequence interpolant $\vec{I}$ of G, there is a corresponding colorable refutation containing the clauses of $\vec{I}$.*

Proof: For simplicity, we only show the case when $N = 2$, where there is only one non-trivial interpolant: I_1. First, we show (10) defines a sequence interpolant. By definition, I_1 is the set of all shared clauses of π colored 1. By definition of coloring, each 1-colored clause is implied by G_1, hence, $G_1 \Rightarrow I_1$. By colorability of π, there is a refutation of $I_1 \wedge G_2$.

Second, for each clause $\alpha \in I_1$, let π_α be a chain refutation of $G_1 \wedge \neg\alpha$, and π_2 be chain refutation of $I_1 \wedge G_2$. The refutation π is obtained by concatenating those refutations. ■

Ordinary chain refutations are exponentially stronger than colorable ones. For example, let k be a natural number and consider the 2-colored CNF $G^k = G_1^k \wedge G_2^k$, where

$$G_1^k = (\bigvee_{i=1}^{k} x_i) \wedge \bigwedge_{i=1}^{k} (x_i \Rightarrow a_i) \wedge (x_i \Rightarrow b_i) \quad G_2 = \bigwedge_{i=1}^{k} (\neg a_i \vee \neg b_i)$$

The CNF interpolant $I_1^k = \text{CNF}(\bigvee_{i=1}^{k}(a_i \wedge b_i))$ is exponential in k. Therefore, a colorable refutation of G is exponential in k as well. Thus, transforming a chain refutation into a colorable one is worst-case exponential. Note that proof-less interpolation techniques such as [18], [19] correspond to colorable chain refutations, and hence, in the worst case are exponentially more expensive than CDCL.

Given a proof π, if `ChainItp` (Alg. 2) returns the interpolant in CNF, then π is colorable. The converse is not true because `ChainItp` picks an arbitrary witness D. Thus, it might not find a colorable witness, even if one exists. We propose two strategies to improve `ChainItp`.

First, we propose to apply UP on line 9 of `ChainItp` *ordered* by the color of the clauses. In the forward-order, UP is first applied to 1-colored clauses, than two 1- and 2-colored clauses, etc. Conversely, the backward-order starts with N-colored clauses. Both strategies increase the number of clauses that are derived within a partition boundary.

Second, we propose a new algorithm to restructure the chain derivation produced by `ConflcitAnalysisTvr` on line 10 of `ChainItp`. The new algorithm, called `ConflictAnalysisClr`, is shown in Alg. 3. It takes a SAT-solver in a conflicting state and a conflict clause, and produces a sequence of chain derivation Π and a new learned clause α. The interpolation step of `ChainItp` (lines 12–19) is then applied to each chain derivation in Π. The main step of the algorithm is done by the supporting procedure, called `Colorize`, shown in Alg. 4.

The algorithms make the following assumptions about the SAT-solver. All clauses are sorted relative to the current assignment so that $\top$-valued literals precede all $\bot$-valued literals. All implied literals are stored in the trail in the implication order. **nil** indicates undefined values (literals and clauses). $\texttt{Value}(q)$ is the value of literal q in the current assignment. $\texttt{Reason}(q)$ is the unique clause that implies the literal q or $\neg q$. $\texttt{Reason}(q) =$ **nil** if q is not implied by any other clause. $\texttt{SetReason}(q, c)$ sets clause c as the reason for q and $\neg q$.

Intuitively, `Colorize` walks the chain derivation from the anchor β_0, and applies only resolutions that are in the same partition as β_0. Clauses from earlier partitions are recursively colorized by attempting to turn them into shared-derived clauses. Clauses from later partitions are ignored. `ConflictAnalysisClr` applies `Colorize` starting from the partition of the anchor, and then as many time as necessary to remove all UP-implied literals from the learned clause. In the worst case, the set Π is linear in the number of clauses in the original chain derivation found by `ConflictAnalysisTvr`.

Algorithm 3: `ConflictAnalysisClr`

Input: A SAT-solver S and a conflict clause *confl*.
Output: A learned clause α and a chain proof Π.

```
1  k ← κ(confl)
2  forever do
3      α ← Colorize(S, confl, k)
4      let T = {q ∈ α | S.Reason(q) = nil}
5      if T = ∅ then break
6      k ← min{κ(q) | q ∈ T}
```

Algorithm 4: `Colorize`

Input: A SAT-solver S, a conflict *confl* and a color k
Output: A learned clause α and a chain proof Π

```
1  p ← nil, α = [ ], β = [ ], W = ∅
2  if S.Value(confl[0]) = ⊤ then
3      p ← confl[0], α.Append(p)
4  forever do
5      if κ(confl) < k then
6          confl ← Colorize(S, confl, κ(confl))
7          S.SetReason(confl[0], confl)
8      β.Append(confl)
9      foreach q ∈ confl do
10         if q = p ∨ q ∈ W ∨ q ∈ α then continue
11         r ← S.Reason(q)
12         if r ≠ nil ∧ κ(r) ≤ k then W ← W ∪ {¬q}
13         else α.Append(q)
14     if W = ∅ then break
15     p ← q ∈ W s.t. q has the largest trail index
16     W ← W \ {p}, confl ← S.Reason(p)
17 if β ≠ [ ] then Π.Append((β ⊢TVR α))
```

VI. Experiments

We have implemented our DRUP-based interpolation framework on top of MiniSAT 2.2. It is available at part of AVY model checker at http://arieg.bitbucket.org/avy. For evaluation, we used two sets of experiments. First, we compared the sizes of the sequence interpolants and the time it takes to extract them for Bounded Model Checking (BMC) problems. Second, we evaluated the framework within our interpolation-based model checker AVY [7]. In both cases, we use benchmarks from HWCCC'13[1]. For baseline, we compare against proof-based interpolation in ABC [25]. Note that we have extended the ABC implementation to sequences in a straight-forward way. However, the comparison with ABC has to be taken with a grain of salt since ABC uses a customized version of an older version of MiniSAT, rewritten in C with some new features back-ported. None-the-less, ABC implementation is the state-of-the-art used by many other hardware model checkers, and we found it to perform well (compared to MiniSAT 2.2).

Fig. 1 shows the sizes of interpolants for BMC problems of depth 20. All problems were given a 180 seconds timeout. In majority of cases, the DRUP-based approaches produce smaller interpolants, measured as number of AIG nodes. Note that for our interpolation algorithm, we conjoin the CNF into the AIG. Clearly, without conjoining this part the interpolants

[1]Benchmarks are available from http://fmv.jku.at/hwmcc13.

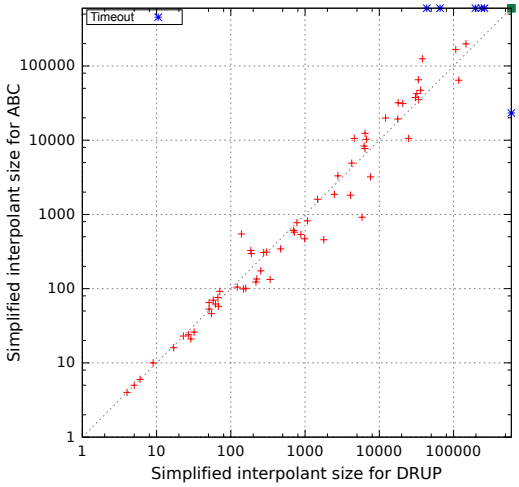

Fig. 1: Comparing sizes (AND gates) of interpolants

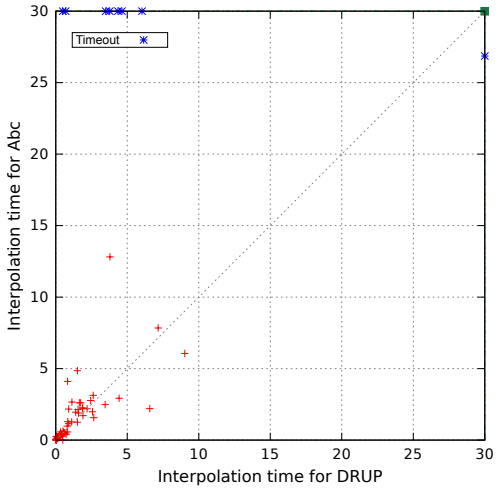

Fig. 2: Comparing time (seconds) to extract interpolants

are even smaller. Careful inspection of the results shows that only two cases were insolvable by DRUP-based methods. On the other hand, 9 cases were not solved by the traditional proof logging solver. This gives us an indication about the strength of a DRUP-based solver, which was apparent in all of our experiments: when the SAT problem becomes hard, the DRUP-based approach outperforms a traditional proof logging solver. Fig. 2 shows the extraction times for the same BMC problems. Note that the extraction time is comparable to the resolution proof based method, but consumes less memory (since the resolution proof is not logged)[2].

Table I analyzes the performance of our model checker AVY when using different interpolation algorithms. This is an important analysis as it shows the affect on the runtime of the model checker and on the depth at which convergence is achieved. First, note that all DRUP-based approaches outperforms our ABC baseline w.r.t. number of solved instances. In addition to that, AVY performs better when using DRUP on the majority of cases. One can note that in most cases, there is a correlation between depth of convergence and performance where lower depth of convergence indicates better runtime.

[2]More comparison charts are at http://arieg.bitbucket.org/avy/drup/plots.

TABLE I: Running time for AVY using different interpolation algorithms. ABC is for ABC's MiniSAT, DRUP is for MiniSAT with DRUP and ordered UP; +Clr adds our colorizing algorithm; and +Pre adds MiniSAT's Pre-processor. 't' stands for time, 'd' for depth of the solution, '–' for time-out or other failure.

Name	ABC		DRUP		DRUP+Clr		DRUP+Pre		DRUP+Pre+Clr	
	t (s)	d	t (s)	d	t (s)	d	t (s)	d	t (s)	d
6s102	203	23	91	24	340	37	175	33	399	37
6s121	–	–	418	50	248	34	–	–	–	–
6s130	136	8	144	9	165	9	192	9	216	9
6s144	–	–	533	25	583	24	668	26	560	22
6s189	622	21	382	21	572	26	396	21	552	23
6s206rb025	66	5	13	3	13	3	15	3	15	3
6s207rb16	46	8	96	8	96	8	127	8	110	8
6s209b1	181	24	106	24	115	24	142	24	157	24
6s215rb0	7	7	4	7	4	7	4	7	4	7
6s216rb0	21	13	12	13	11	13	13	13	13	13
6s218b1246	588	9	283	9	272	9	293	9	281	9
6s271rb045	371	11	256	10	262	10	–	–	–	–
6s273b37	162	20	216	20	217	20	277	20	284	20
6s275rb253	4	6	7	6	7	6	8	6	10	6
6s276rb318	19	10	11	10	11	10	15	10	16	10
6s277rb342	18	10	15	13	10	10	13	10	22	13
6s282b15	103	18	99	25	106	19	184	17	209	17
6s288r	–	–	376	24	338	22	468	23	444	22
6s289rb00529	77	7	38	7	37	7	38	7	37	7
6s291rb18	517	78	341	74	283	73	–	–	717	73
6s305rb069	270	18	139	18	128	18	133	18	136	18
6s306rb03	219	17	58	13	56	13	58	13	59	13
6s307rb06	127	13	86	13	90	13	102	13	109	13
6s311rb1	69	2	19	2	18	2	20	2	20	2
6s326rb02	34	11	14	11	15	11	17	11	17	11
6s327rb10	25	9	11	9	11	9	12	9	12	9
6s330rb11	10	3	5	3	4	3	5	3	5	3
6s335rb60	2	4	1	4	1	4	1	4	1	4
6s343b31	–	–	–	–	–	–	332	15	503	15
6s349rb12	185	13	143	15	142	15	158	15	169	15
6s364rb03158	519	2	198	2	198	2	191	2	188	2
6s372rb31	358	29	322	30	162	21	295	29	276	26
6s374b029	467	9	264	9	258	9	264	9	256	9
6s380b129	226	20	109	20	109	20	131	20	122	20
6s384rb194	–	–	–	–	–	–	786	22	868	30
6s385rb444	441	13	237	12	257	13	218	12	203	12
6s386rb07	–	–	871	13	868	13	855	13	828	13
6s388b07	0	0	0	0	0	0	0	0	0	0
6s389b11	6	4	3	4	3	4	3	4	3	4
6s38	341	14	301	15	296	13	624	19	347	14
6s403rb0609	17	5	11	5	12	5	14	5	14	5
6s404rb4	55	4	45	4	65	5	69	4	78	4
6s405rb611	85	6	53	6	54	6	58	6	65	6
6s406rb111	735	16	521	16	612	16	544	16	662	17
6s407rb296	417	12	354	12	360	12	378	12	405	12
6s408rb191	264	8	452	8	420	8	340	8	371	8
6s410rb043	193	9	150	9	156	9	275	10	273	10
6s9	166	10	194	9	219	9	309	9	221	9
SOLVED	42		46		46		45		46	

Also note that this experiment confirms the results of the above figures which show that interpolation time is comparable with a proof-logging SAT solver and that sizes are in favor of DRUP.

Another important analysis is the effect *Colorizing* has on AVY's performance. Clearly, using colorize results in different interpolants. We can see from the results that there are cases where this results in better convergence depths and thus better performance. Note that using this feature is more demanding than simply extracting an interpolant since it restructures local chain derivations. Even though, when the convergence depth is similar the performance degradation due to the extra computation is small. It is important to note that colorizing results in many shared-derivable leaves, which means that the CNF component of the interpolant is meaningful. Currently, we did not make any special use of the CNF component and we leave this option for future research and exploration.

Finally, in Table II, we show the number of shred-derivable leaves, i.e. number of clauses in the CNF component of the interpolant computed by our method. Recall that DRUP is used

TABLE II: Number of shared-derivable leaves using our interpolation algorithm when solving BMC problems using bound 20. Algorithm names are as in Table I.

Name	DRUP	DRUP+Clr	DRUP+Pre	DRUP+Pre+Clr
6s102	0	73	0	257
6s119	0	976	0	0
6s122	0	223	0	216
6s152	0	449	0	217
6s188	0	651	0	521
6s196	0	648	0	642
6s276rb318	0	230	0	122
6s27	0	507	0	572
6s282b15	0	1684	0	270
6s291rb18	0	420	24	177
6s292rb024	0	1043	0	577
6s302rb09	0	1257	0	369
6s309b046	0	641	0	669
6s310r	0	1334	1	810
6s351rb02	0	6956	0	6910
6s384rb194	0	1144	0	408
6s44	0	1701	0	1188
6s50	0	617	0	166
6s7	0	1372	1	860
6s8	0	1123	1	615

with ordered UP while DRUP+Clr is used with ordered UP and colorize. Here too, we use fixed bound BMC problems. It is clear that our colorizing algorithm is very effective in finding a large number of clauses. While we present only a selected subset, this trend holds in all our experiments.

Note that while the underlying model checking algorithm AVY did not make a special use of the CNF component, we believe that specialized usage of the CNF component will result in better performance [5], [7], [12].

VII. Related Work

To our knowledge, this paper is the first to present and *evaluate* a DRUP-based interpolation framework. Moreover, we introduce a novel algorithm that computes a *sequence interpolant* partially in CNF. Finally, our restructuring algorithm is not based on pivot reordering as in previous works, but tries to keep resolution steps within a given partition (colorizing). We have already discussed proof-based and proof-less interpolation methods in Sec. I, and clausal proofs in Sec. III. Thus, in this section, we only focus on proof restructuring for CNF.

Many works deal with generating better interpolants, either using new interpolation algorithms or by proof restructuring. Our work is a synergy of these two approaches. [11] and [13] suggest local transformation rules that are based on pivots reordering to get CNF interpolants. Rollini et al. [13] also suggest a compression of a resolution proof as a pre-processing step. Unlike our work, they rely on explicit resolution proofs. Furthermore, our restructuring does not rely on pivot reordering and supports sequence interpolation natively.

Our interpolation algorithm identifies the CNF component of an interpolant even if the interpolant itself is not in CNF. Vizel at al. [12] introduce an interpolation procedure that also produces (near) interpolants in CNF. However, unlike [12], our framework does not rely on explicit resolution proofs and produced complete interpolants. We leave extending [12] to DRUP-proofs for future work.

VIII. Conclusion

In this paper, we introduce a DRUP-based interpolation framework. We show how DRUP-proofs can be trimmed and restructured for interpolation. We develop a novel interpolation algorithm that computes interpolants partially in CNF. Furthermore, we show how DRUP-proofs can be locally restructured to maximize the size of the CNF component without exponentially increasing the proof. Based on previous works [5], [7], [12], we believe that getting a CNF component for an interpolant is beneficial for the underlying model checking algorithm. Our framework is implemented in MiniSAT and is publicly available. Our experiments show that the framework is very effective in the context of both bounded and unbounded model checking applications.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *TACAS*, 1999, pp. 193–207.

[2] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *CAV*, 2003, pp. 1–13.

[3] ——, "Lazy abstraction with interpolants," in *CAV*, 2006, pp. 123–136.

[4] Y. Vizel and O. Grumberg, "Interpolation-sequence based model checking," in *FMCAD*, 2009, pp. 1–8.

[5] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *VMCAI*, 2011, pp. 70–87.

[6] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, 2011, pp. 125–134.

[7] Y. Vizel and A. Gurfinkel, "Interpolating property directed reachability," in *CAV*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 260–276.

[8] P. Pudlák, "Lower bounds for resolution and cutting plane proofs and monotone computations," *J. Symb. Log.*, vol. 62, no. 3, 1997.

[9] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher, "Interpolant strength," in *VMCAI*, 2010, pp. 129–145.

[10] G. Weissenbacher, "Interpolant strength revisited," in *SAT*, 2012.

[11] R. Jhala and K. L. McMillan, "Interpolant-Based Transition Relation Approximation," in *CAV*, 2005, pp. 39–51.

[12] Y. Vizel, V. Ryvchin, and A. Nadel, "Efficient generation of small interpolants in cnf," in *CAV*, 2013, pp. 330–346.

[13] S. F. Rollini, R. Bruttomesso, N. Sharygina, and A. Tsitovich, "Resolution proof transformation for compression and interpolation," *CoRR*, vol. abs/1307.2028, 2013.

[14] A. Gurfinkel, S. F. Rollini, and N. Sharygina, "Interpolation properties and sat-based model checking," in *ATVA*, 2013, pp. 255–271.

[15] A. Van Gelder, "Producing and verifying extremely large propositional refutations - have your cake and eat it too," *Ann. Math. Artif. Intell.*, vol. 65, no. 4, pp. 329–372, 2012.

[16] M. Heule, W. A. Hunt, Jr, and N. Wetzler, "Trimming while checking clausal proofs," in *FMCAD*, 2013, pp. 181–188.

[17] S. F. Rollini, L. Alt, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina, "PeRIPLO: A Framework for Producing Effective Interpolants in SAT-Based Software Verification," in *LPAR*, 2013, pp. 683–693.

[18] H. Chockler, A. Ivrii, and A. Matsliah, "Computing interpolants without proofs," in *Haifa Verification Conference*, 2012, pp. 72–85.

[19] S. Bayless, C. G. Val, T. Ball, H. H. Hoos, and A. J. Hu, "Efficient modular SAT solving for IC3," in *FMCAD*, 2013, pp. 149–156.

[20] E. I. Goldberg and Y. Novikov, "Verification of Proofs of Unsatisfiability for CNF Formulas," in *DATE*, 2003, pp. 10 886–10 891.

[21] P. Beame, H. A. Kautz, and A. Sabharwal, "Towards understanding and harnessing the potential of clause learning," *J. Artif. Intell. Res. (JAIR)*, vol. 22, pp. 319–351, 2004.

[22] A. Biere, "PicoSAT Essentials," *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.

[23] W. Craig, "Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem," *J. Symb. Log.*, vol. 22, no. 3, pp. 250–268, 1957.

[24] J. P. M. Silva, I. Lynce, and S. Malik, "Conflict-Driven Clause Learning SAT Solvers," in *Handbook of Satisfiability*, 2009, pp. 131–153.

[25] R. K. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *CAV*, 2010, pp. 24–40.