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 unsatisfiable 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 weaknesses 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 interpolation, 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 effect 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) \cite{bmc} is reduced to a single satisfiability 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 \( \pi \) of bounded safety, and (c) attempting to generalize \( \pi \) to an inductive invariant. The bounded safety proof \( \pi \) 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 \( G \equiv A \land B \) 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 \cite{bmc}. 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, \( G \equiv G_1 \land \cdots \land G_N \) 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 \( G_i \land \cdots \land G_i \) and \( G_{i+1} \land \cdots \land G_N \). 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 \( \bigvee_{1 \leq j \leq i} I_j \). Most SAT-based model checking algorithms (e.g., \cite{bmc}–\cite{bmc}) 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 \cite{bmc}. They are simple to implement and produce interpolants that are linear in the size of the proof. Their variations (for strength \cite{bmc}, \cite{bmc}) structure \cite{bmc} and size \cite{bmc} and model checking specific properties (e.g., \cite{bmc}, \cite{bmc}) are widely studied. However, they require a SAT-solver to log the resolution proof. While this is not technically difficult \cite{bmc}, it significantly increases the memory usage of the solver \cite{bmc}. Furthermore, it appears that combining proof-logging and common pre-processing is difficult. Most solvers (e.g., \cite{bmc}) treat proof-logging and pre-processing as mutually exclusive.

Alternatively, interpolants can be constructed by partitioning the clauses of \( G \equiv A \land B \) 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 \cite{bmc} (together with many other improvements), has been applied for interpolation by Chockler et al. \cite{bmc}, and has been further refined by Bayless et al. \cite{bmc} 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 \cite{bmc} 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. \cite{bmc} 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
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 [10], 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 \( p \land g \) 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 \( \ell \) 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 \( \Box \) to denote the empty clause, \( Var(\alpha) \) for variables of a clause \( \alpha \), and \( Var(G) \) for variables of a set of clauses \( G \).

The resolution rule states that given clauses \( \alpha_1 = \beta_1 \lor v \) and \( \alpha_2 = \beta_2 \lor \neg v \), where \( \beta_1 \) and \( \beta_2 \) are clauses and \( v \) and \( \neg v \) are literals, one can derive the clause \( \alpha_3 = \beta_1 \lor \beta_2 \). Application of the resolution rule is denoted by \( \alpha_1, \alpha_2 \vdash_{\text{RES}} \alpha_3 \), 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 \( \alpha \) from a CNF formula \( G \) is a sequence \( \pi = (\alpha_1, \alpha_2, \ldots, \alpha_n, \alpha) \), where each clause \( \alpha_k \) is either an initial clause of \( G \) or is derived by applying the resolution rule to clauses \( \alpha_i, \alpha_j \) with \( i, j < k \). A resolution derivation of the empty clause \( \Box \) from \( G \) is called a refutation or a proof, and shows that \( G \) is unsatisfiable.

A resolution derivation \( (\alpha_1, \ldots, \alpha_n) \) of \( G \) is trivial [21] if all variables resolved upon are distinct, and each \( \alpha_i \), for \( i \geq 3 \), is either an initial clause or is derived by resolving \( \alpha_{i-1} \) with an initial clause. It is convenient to capture a trivial resolution derivation by a rule. A chain resolution rule, written \( \alpha_1, \ldots, \alpha_k \vdash_{\text{TVR}} \alpha \), states that \( \alpha \) can be derived from \( \alpha_1, \ldots, \alpha_k \) by trivial resolution derivation. We call \( \alpha_1, \ldots, \alpha_k \) the chain and \( \alpha_1 - \) the anchor, and variables \( \vec{x} = (x_1, \ldots, x_k) \) the chain pivots. Without loss of generality, we assume that the chain and chain pivots are resolved in the order given. That is, first \( \alpha_1 \) is resolved with \( \alpha_2 \) on \( x_1 \), then the resolvent is resolved with \( \alpha_3 \) on \( x_2 \), etc. A chain derivation is a sequence \( \pi = (\alpha_1, \ldots, \alpha_n) \) where each \( \alpha_k \) is either an initial clause or is derived by chain resolution from preceding clauses. A derivation witness of a chain derivation \( \pi \) is a total function \( D \) from clauses of \( \pi \) to sub-sequences of \( \pi \) such that

\[
D(\alpha) = \{ \] \Rightarrow \alpha \text{ is initial} \quad D(\alpha) \neq \{ \Big) \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 \( A \land B \) [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 \( (G, \kappa) \) of a CNF formula \( G \) and a coloring function \( \kappa : G \to [1, \ldots, N] \) that assigns to every clause \( \alpha \in G \) a color between 1 and \( N \). We omit the coloring function \( \kappa \) when it is clear from the context or irrelevant and write \( G \) for \( (G, \kappa) \). For a colored CNF \( (G, \kappa) \), 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:

\[
\kappa_\downarrow(v) = \min \{ i \mid \exists \alpha \in G_i \land v \in \alpha \} \quad (3)
\]

\[
\kappa_\uparrow(v) = \max \{ i \mid \exists \alpha \in G_i \land v \in \alpha \} \quad (4)
\]

A variable \( v \) is called local (to partition \( i \)) if \( \kappa_\downarrow(v) = \kappa_\uparrow(v) = i \), and shared otherwise. A clause \( \alpha \) 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 \( \pi \) of a colored
CNF $(G, \kappa)$ and a derivation witness $D$ of $\pi$, we define the maximum color for the clauses of $\pi$ inductively as follows:

\[
\kappa_i(\alpha) = \begin{cases} 
\kappa(\alpha) & \text{if } \alpha \in G \\
\max\{\kappa_i(\beta) | \beta \in D(\alpha)\} & \text{otherwise}
\end{cases}
\]

Minimum color $\kappa_i(\alpha)$ is defined similarly.

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

\[
I_{i-1} \land G_i \Rightarrow I_i \
\forall v \in \text{Var}(I_i) \cdot \kappa_i(v) = i \land \kappa_i(v) = i + 1
\]

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:

- **UnitPropagate** 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 $\pi \equiv (\{d_0, d_0\}, \ldots, (d_n, d_n) \equiv (\square, \bot))$, where each $d_k$ is a Boolean flag indicating whether the clause is deleted, and $\alpha_k$ is either an initial clause or is derived by chain resolution from the set of $k$-active clauses $\{\alpha_j | j < k \land d_j = \bot \land \forall j < i < k \cdot \alpha_i \neq \alpha_j\}$. Validation of DRUP-proofs is efficient because validation of a clause $\alpha_k$ 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 $\pi$. The output is a chain derivation $\pi$ such that all clauses of $\pi$ participate in a derivation of the empty clause. In the terminology of Heule et al., all clauses of $\pi$ are core. The algorithm maintains a set $C$ of core clauses. It walks the input DRUP-proof $\pi$ backwards. Deleted clauses are revived (line 5). If the current clause $\alpha_i$ is on the trail, **UndoTrailCore** is used to pop the literals of the trail up to and including the literal whose antecedent is $\alpha_i$. In the process, antecedents of any core literal on the trail are marked core as well. Next, $\alpha_i$ 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 $\pi$ 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, $\pi$ 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 $(G, \kappa)$ 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 $V_j = \text{Var}(G_j) \cap \text{Var}(G_{j+1})$. For a clause $\alpha \in G$, we write $\alpha_{\lfloor k \rfloor}$ for a clause obtained from $\alpha$ by removing all variables $v$ with color less than $k$ ($\kappa(v) < k$) or greater than $l$ ($\kappa(v) > l$). We write $\alpha_{\lfloor l \rfloor}$ for $\alpha_{\lfloor l \rfloor}$ and $\alpha_{\lfloor k \rfloor}$ for $\alpha_{\lfloor k \rfloor}$. Recall that a clause $\alpha$ is shared w.r.t. $j$ if $\text{Var}(\alpha) \subseteq V_j$ and $\kappa(\alpha) = j$. 
Our procedure, called \texttt{ChainItp}, is shown in Alg.\ref{alg:chainitp}. The inputs are a \(N\)-colored CNF \((G,\kappa)\) and a (trimmed) chain derivation \(\pi\). The output is a sequence interpolant \(I_0, \ldots, I_N\). \texttt{ChainItp} walks \(\pi\) forward from \(\alpha_0\) to \(\alpha_n\) and computes partial interpolants for each partition (or color) separately. For partition \(g\) and a clause \(\alpha_j\), a partial interpolant is a conjunction of a pair of formulas \(p_l(\alpha_j)\land q_{\beta_j}; q_{\beta_j}\) contains the CNF part of the interpolant, and \(p_l(\alpha_j)\) contains the rest. The final interpolant is obtained as a partial interpolant of the empty clause \(\alpha_0 \equiv \top\).

For a fixed color \(k\), we partition the clauses of \(\pi\) 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 interleaved using helper functions \texttt{Leaf} and \texttt{Tvr}, respectively. Before going into detail, let us introduce the following notion:

\begin{definition}
Let \((G,\kappa)\) be an \(N\)-colored striped CNF formula, \(\pi\) a chain refutation of \(G\), \(D\) a derivation witness for \(\pi\), and \(k\) a natural number \(1 \leq k \leq N\). A shared leaf \(\alpha \in \pi\) is \(\textit{shared-derivable}\) \(w.r.t.\) \(k\) and \(D\) if for all \(\beta \in D(\alpha), \kappa_k(\beta) = k\) or \(\beta\) is shared-derivable \(w.r.t.\) \(k \land 1 - D\). Clearly, for initial shared clauses, this definition holds trivially. Intuitively, \(\alpha\) 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 \(G_1 \land G_2 \land G_3\). 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.

\begin{lemma}
Let \((G,\kappa)\) be an \(N\)-colored striped CNF formula. Given a chain derivation \(\pi\), let \(D\) be a derivation witness of \(\pi\).

Let \((g_0 = \top, g_1, \ldots, g_N)\) be a sequence such that \(g_i\) is a CNF containing all shared-derivable clauses \(w.r.t.\) a color \(i\) and \(D\), then \(g_{i-1} \land G_i \Rightarrow g_i\) 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 \texttt{Leaf} and \texttt{Tvr}. The function \texttt{Leaf} is applied to initial clauses (line \(1\)) and to derived leaf clauses (line \(15\)). The input is a clause \(\alpha\), a color \(j\) and a derivation witness \(D\). The output is a pair \((p, q)\) such that \(p \land q\) is a partial interpolant of \(\alpha\) for color \(j\) and \(g\) is in CNF. It works according to the following rules:

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

The function \texttt{Tvr} is applied to derived clauses. The input is a clause \(\alpha\), a corresponding chain derivation \(\beta = \text{ChainDRUP}^{\top}\alpha\), and a color \(j\). The chain \(\beta = (\beta_1, \ldots, \beta_N)\) is obtained by UP and conflict analysis (lines \(8\)\(10\)) as described in Section \(\text{III}\). The output is a formula \(q_{\beta}\), where

\[q_{\beta} = \begin{cases} p_j(\beta_0) & \text{if } l = 0 \\ q_{\beta-1} \land p_j(\beta_l) & \text{ow} \end{cases}\]

That is, \texttt{Tvr} walks up the chain \(\beta\), and at each resolution step, either conjoins or disjoins the partial interpolants of the chain clauses. \texttt{Tvr} is effectively a direct extension of the interpolation rules of \cite{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 \texttt{Leaf}. We only mention it in Algorithm \ref{alg:chainitp} 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.

\begin{definition}
Given an unsatisfiable \(N\)-colored striped CNF \((G,\kappa)\) and a chain derivation \(\pi\). A sequence of partial interpolants \((\top, p_1, \ldots, p_{N-1}, \bot)\) and a set of CNF formulas \(\{g_j\}_{j=1}^{N-1}\) are valid iff for every \(1 \leq k \leq N\), and for every \(\alpha \in \pi\), \((p_k(\alpha) \land g_k \land G_{k+1}) \implies (p_{k+1}(\alpha) \lor \alpha\big|_{\geq k+1}) \land g_{k+1}\).

Note that a valid partial interpolant sequence results in a valid sequence interpolant. We show that the partial interpolants of \texttt{ChainItp} satisfy validity requirement of Def \cite{2}.

\begin{theorem}
Given an \(N\)-colored striped CNF \((G,\kappa)\) and a chain derivation \(\pi\), the sequence of partial interpolants \((\top, p_1, \ldots, p_{N-1}, \bot)\) and the set of CNF formulas \(\{g_j\}_{j=1}^{N-1}\) computed by \texttt{ChainItp} are valid.

\textbf{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, \texttt{ChainItp} is a straightforward extension of McMillan’s procedure \cite{2} to chain resolution. We use \(q_j(\alpha)\) to denote the partial interpolant of \cite{2}.
Algorithm 2: ChainItp

Input: A SAT-solver instance $S$, colored CNF $(G, \kappa)$,
$\kappa : G \rightarrow [1..N]$, and a chain derivation
$\pi = (a_0, \ldots, a_n \equiv \bot)$
Output: An interpolation sequence
$(T \equiv I_0, I_1, \ldots, I_N \equiv \bot)$

1. for $i = 0$ to $n$ do
   2. if $a_i \in G$ then
      3. for $j = 1$ to $N - 1$ do
         4. $(p_j(a_i), g) \leftarrow \text{Leaf}(a_i, j)$
         5. $g_j \leftarrow g_j \land g$
   else
      6. $\text{S.UnitPropagate}(), \text{S.SaveTrail}()$
      7. $\text{S.Enqueue}(\neg a_i)$
      8. $\beta_0 = \text{S.UnitPropagate}()$
      9. $\bar{\beta} = \text{S.ConflictAnalysisTvr}(\beta_0, a_i)$
         /* $\bar{\beta}$ is a subsequence of $\pi$ s.t. $\bar{\beta} \uparrow \text{Tvr} a_i$ */
      10. $D(a_i) \leftarrow \bar{\beta}$
      11. $\kappa(a_i) \leftarrow \max \{\kappa(c) \mid c \in \bar{\beta}\}$
      12. for $j = 1$ to $N - 1$ do
         13. if $\kappa(a_i) \leq j$ then
            14. $(p_j(a_i), g) \leftarrow \text{Leaf}(a_i, j)$
            15. $g_j \leftarrow g_j \land g$
         else
            16. $(p_j(a_i)) \leftarrow \text{Tvr}(\bar{\beta}, a_i, j)$
      17. $\text{S.RestoreTrail}()$
      18. $\text{S.Revive}(a_i)$
   19. $I_0 \leftarrow T, I_N \leftarrow \bot$
20. for $j = 1$ to $N - 1$ do $I_j \leftarrow p_j(a_n) \land g_j$

The proof is by induction on the graph induced by $\pi$ and $D$. The base case follows from [14] since for an initial clause $\alpha$ and $p_j(\alpha) \land g_j = g_j(\alpha)$. 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:

$$p_1(c_1) \land g_1 \land G_2 \Rightarrow (p_2(c_2) \lor c_1 \lor g_1) \lor g_2 \quad (6)$$
$$p_1(c_2) \lor g_1 \land G_2 \Rightarrow (p_2(c_2) \lor c_2 \lor g_1) \lor g_2 \quad (7)$$

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:

$$p_1(c_1) \land g_1 \land G_2 \Rightarrow (c_1 \lor g_1 \lor G_2) \land g_2 \quad (8)$$
$$p_1(c_2) \lor g_1 \land G_2 \Rightarrow (c_2 \lor g_1 \lor G_2) \land g_2 \quad (9)$$

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

Case 2, $c$ is not shared-derivable. We need to show that $(p_1(c) \land g_1) \land G_2 \Rightarrow (c_1 \lor g_1) \land g_2$. Since $c$ is not shared-derivable both $g_1$ and $g_2$ are unchanged. Assume that $v \notin \text{Var}(G_3)$, then $p_1(c) = p_1(c_1) \lor p_1(c_2)$. Assume, to the contrary, that $p_1(c) \land g_1 \land G_2 \Rightarrow c_2 \lor g_2$ does not hold. Then, there is an assignment s.t. $(p_1(c) \land g_1) \land G_2$ evaluates to $T$ while $c_2 \lor g_2$ evaluates to $\bot$. From Lemma [2] we know that $g_2$ evaluates to $T$, therefore, $c_2$ is $\bot$. W.l.o.g. assume that under this assignment $p_1(c_1)$ evaluates to $T$ as well. Due to our assumption that $c_2 \lor g_2$ evaluates to $\bot$, $v$ must evaluate to $T$, but, since $v \in c_1 \lor c_2$, it must also be part of $c_2$. Thus, indicating that $c_2 \lor g_2$ evaluates to $T$, 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 colorability.

Let $(G, \kappa)$ be a striped $N$-colored CNF, $\pi$ a chain refutation of $G$, and $D$ a derivation witness for $\pi$. 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 $0 \leq i \leq n$, $\kappa_i(\beta_i) = \kappa_i(\beta_i) = \kappa_i(\alpha)$ or $\kappa_i(\beta_i) < \kappa_i(\alpha)$ and $\beta_i$ is shared. A chain refutation $\pi$ is colorable if there exists a colored refutation witness for it.

Colorable refutations induce a simple interpolation procedure. Let $\pi = (a_0, \ldots, a_n)$ be a colorable chain refutation with $N$ colors. Then, the sequence $\bar{\pi} \equiv (I_0 \equiv \top, \ldots, I_N \equiv \bot)$ defined as follows:

$$I_i = \{\alpha \in \pi \mid \kappa_i(\alpha) = i \land \alpha \text{ is shared}\} \quad (10)$$

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

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

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 $\pi$ colored 1. By definition of coloring, each 1-colored clause is implied by $G_1$, hence, $G_1 \Rightarrow I_1$. By colorability of $\pi$, there is a refutation of $I_1 \land G_2$. 


Second, for each clause \( \alpha \in I_1 \), let \( \pi_\alpha \) be a chain refutation of \( G_1 \land \neg \alpha \), and \( \pi_\beta \) be chain refutation of \( I_1 \land G_2 \). The refutation \( \pi \) 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 \( G^k = G^k_1 \land G^k_2 \), where
\[
G^k_1 = \bigwedge_{i=1}^k (x_i) \land \bigwedge_{i=1}^k (x_i \Rightarrow a_i) \land (x_i \Rightarrow b_i) \quad G^k_2 = \bigwedge_{i=1}^k (\neg a_i \lor \neg b_i)
\]

The CNF interpolant \( I^k = \text{CNF}(V_{i=1}^k (a_i \lor b_i)) \) 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 proofless interpolation techniques such as \cite{18}, \cite{19} correspond to colorable ones. For example, let \( 2 \)-colored CNF \( (\neg a \lor \neg b) \land (\neg c \lor \neg d) \) be a natural number and \( k \) in \( G \). Given a proof \( \pi \), if ChainItp (Alg. 2) returns the interpolant in CNF, then \( \pi \) 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 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 ConflictAnalysisTvr on line [10] of ChainItp. The new algorithm, called ConflictAnalysisClr, is shown in Alg. 3. It takes a SAT-solver \( S \) and a conflict \( confl \) and a color \( k \), and produces a sequence of chain derivation II and a new learned clause \( \alpha \). The interpolation step of ChainItp (lines [12],[19]) is then applied to each chain derivation in II. The main step of the algorithm is done by the supporting procedure, called Colorize, shown in Alg. 4.

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

Intuitively, Colorize walks the chain derivation from the anchor \( \beta_0 \) and applies only resolutions that are in the same partition as \( \beta_0 \). 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 II is linear in the number of clauses in the original chain derivation found by ConflictAnalysisTvr.

### 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 \url{http://arieg.bitbucket.org/avy}. For evaluation, we used two sets of experiments. First, we compared the sizes of the sequence of 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 \textsc{Avy} \cite{7}. In both cases, we use benchmarks from HWCCC'13 \cite{7}. For baseline, we compare against proof-based interpolation in ABC \cite{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 conjoinning this part the interpolants

---

1Benchmark marks are available from \url{http://fmv.jku.at/hwcc13}
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 logger. 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).

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 outperform 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.

Table II shows 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

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 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>ABC</th>
<th>DRUP</th>
<th>DRUP+Clr</th>
<th>DRUP+Pre</th>
<th>DRUP+Pre+Clr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t(s)</td>
<td>t(s)</td>
<td>t(s)</td>
<td>t(s)</td>
<td>t(s)</td>
</tr>
<tr>
<td>6s121</td>
<td>20</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>6s122</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>6s123</td>
<td>11</td>
<td>8</td>
<td>14</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>6s124</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6s125</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6s126</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6s127</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>6s128</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>6s129</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>6s130</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>DRUP</th>
<th>DRUP+Clr</th>
<th>DRUP+Pre</th>
<th>DRUP+Pre+Clr</th>
</tr>
</thead>
<tbody>
<tr>
<td>6s102</td>
<td>0</td>
<td>73</td>
<td>0</td>
<td>257</td>
</tr>
<tr>
<td>6s119</td>
<td>0</td>
<td>976</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6s122</td>
<td>0</td>
<td>223</td>
<td>0</td>
<td>223</td>
</tr>
<tr>
<td>6s152</td>
<td>0</td>
<td>449</td>
<td>0</td>
<td>217</td>
</tr>
<tr>
<td>6s188</td>
<td>0</td>
<td>681</td>
<td>0</td>
<td>321</td>
</tr>
<tr>
<td>6s196</td>
<td>0</td>
<td>648</td>
<td>0</td>
<td>642</td>
</tr>
<tr>
<td>6s26fb318</td>
<td>0</td>
<td>230</td>
<td>0</td>
<td>122</td>
</tr>
<tr>
<td>6s27</td>
<td>0</td>
<td>507</td>
<td>0</td>
<td>572</td>
</tr>
<tr>
<td>6s282b15</td>
<td>0</td>
<td>1684</td>
<td>0</td>
<td>270</td>
</tr>
<tr>
<td>6s291b18</td>
<td>0</td>
<td>429</td>
<td>24</td>
<td>177</td>
</tr>
<tr>
<td>6s292b024</td>
<td>0</td>
<td>1043</td>
<td>0</td>
<td>577</td>
</tr>
<tr>
<td>6s322b09</td>
<td>0</td>
<td>1259</td>
<td>0</td>
<td>369</td>
</tr>
<tr>
<td>6s309b046</td>
<td>0</td>
<td>641</td>
<td>0</td>
<td>669</td>
</tr>
<tr>
<td>6s310</td>
<td>0</td>
<td>1334</td>
<td>1</td>
<td>810</td>
</tr>
<tr>
<td>6s315b02</td>
<td>0</td>
<td>6956</td>
<td>0</td>
<td>6910</td>
</tr>
<tr>
<td>6s384b194</td>
<td>0</td>
<td>1144</td>
<td>0</td>
<td>408</td>
</tr>
<tr>
<td>6s44</td>
<td>0</td>
<td>1701</td>
<td>0</td>
<td>1188</td>
</tr>
<tr>
<td>6s50</td>
<td>0</td>
<td>617</td>
<td>0</td>
<td>175</td>
</tr>
<tr>
<td>6s57</td>
<td>0</td>
<td>1345</td>
<td>0</td>
<td>860</td>
</tr>
<tr>
<td>6s68</td>
<td>0</td>
<td>1123</td>
<td>1</td>
<td>615</td>
</tr>
</tbody>
</table>

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 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