Proofs in conflict-driven theory combination
Maria Paola Bonacina, Stéphane Graham-Lengrand, Natarajan Shankar

To cite this version:
Maria Paola Bonacina, Stéphane Graham-Lengrand, Natarajan Shankar. Proofs in conflict-driven theory combination. J. Andronick and A. Felty. Proceedings of the 7th International Conference on Certified Programs and Proofs (CPP’18), Jan 2018, Los Angeles, United States. ACM Press, 18, <10.1145/3167096>. <hal-01935595>

HAL Id: hal-01935595
https://hal.archives-ouvertes.fr/hal-01935595
Submitted on 27 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Abstract
Search-based satisfiability procedures try to construct a model of the input formula by simultaneously proposing candidate models and deriving new formulæ implied by the input. When the formulæ are satisfiable, these procedures generate a model as a witness. Dually, it is desirable to have a proof when the formulæ are unsatisfiable. Conflict-driven procedures perform nontrivial inferences only when resolving conflicts between the formulæ and assignments representing the candidate model. CDSAT (Conflict-Driven SATisfiability) is a method for conflict-driven reasoning in combinations of theories. It combines solvers for individual theories as theory modules within a solver for the union of the theories. In this paper we endow CDSAT with lemma learning and proof generation. For the latter, we present two techniques. The first one produces proof objects in memory: it assumes that all theory modules produce proof objects and it accommodates multiple proof formats. The second technique adapts the LCF approach to proofs from interactive theorem proving to conflict-driven SMT-solving and theory combination, by defining a small kernel of reasoning primitives that guarantees that CDSAT proofs are correct by construction.

CCS Concepts  • Theory of computation → Automated reasoning:

Keywords Proof generation, Lemma learning, Theory combination, Satisfiability modulo assignment

1 Introduction
The satisfiability problem is one of checking if a given formula has a model. Propositional satisfiability (SAT) is the problem of finding an assignment of truth values to variables that satisfies the input propositional formula, typically given as a set of clauses. Many SAT solvers employ a conflict-driven search strategy, known as Conflict-Driven Clause Learning (CDCL), in which the solver extends a partial assignment until it satisfies the formula, or a conflict arises because the assignment falsifies the formula. Nontrivial inference steps are performed in response to a conflict to roll back the partial assignment and direct the search elsewhere [27]. This conflict-driven style inspired the design of several solvers for quantifier-free fragments of arithmetic [12, 19, 23–25, 29, 35, 36]. These conflict-driven theory solvers decide the satisfiability of sets of literals in the theory. The problem of deciding the satisfiability of sets of clauses in a theory is known as Satisfiability Modulo a Theory (SMT). MCSAT, for Model Constructing SATisfiability, is an approach to conflict-driven SMT [14]. It integrates a CDCL-based SAT-solver and a conflict-driven theory solver equipped with an inference system to detect and explain conflicts in the theory. In MCSAT, both Boolean and first-order variables are given assignments in a trail representing the candidate model as a partial assignment. First-order variables are assigned constant symbols representing individuals of the corresponding sort in a model’s domain. For example, integer variables are assigned integer constants, and bit vector variables are assigned constant bit vectors. MCSAT allows the introduction of new (i.e., non-input) terms in lemmas as long as these terms are drawn from a finite basis. The MCSAT approach has been shown to be quite versatile and efficient [21, 22, 38].

Most problems involve more than one theory, and MCSAT is not a combination calculus in general. In prior work, we designed a method called Conflict-Driven SATisfiability (CDSAT) to solve in a conflict-driven manner satisfiability problems in a generic combination of disjoint theories [6, 7]. CDSAT combines multiple theory solvers, all or some of...
which are conflict-driven, into a conflict-driven solver for the union of the theories. More precisely, CDSAT combines theory inference systems, called theory modules: the only relevant component of a conflict-driven theory solver is its inference system, since the conflict-driven search is performed by CDSAT for all theories. A non-conflict-driven solver is abstracted into a theory module whose only inference rule invokes the solver to detect unsatisfiability in the theory.

In CDSAT, a common trail contains assignments to both Boolean and first-order terms. Some assignments are decisions, that is, guesses, and others are propagated because they are entailed by the formula and prior assignments in the trail. Theory modules work on this trail to detect conflicts with any theory-specific input formula. Conflict resolution rules work to roll back the trail to a prior decision level. The CDSAT transition system is sound, terminating, and complete, relative to the theory modules and a global finite basis for all theories. CDSAT generalizes CDCL, MCSAT, and the equality sharing (Nelson-Oppen) method for combination of theories [26, 31]: CDSAT reduces to CDCL, if propositional logic is the only theory, to MCSAT, if the combination features propositional logic and another theory with conflict-driven solver, and to equality sharing, if no theory in the combination has a conflict-driven solver [7].

In this paper we extend CDSAT with lemma learning (Section 3) and proof generation (Sections 4 and 5). Lemmas are formulae entailed by the input formula. Learning lemmas from conflicts is a key feature of conflict-driven reasoning engines, because conflict-driven lemmatization immediately thwarts any attempt to repeat a failed search path (e.g., [27]). Proof generating inference systems [33] are important because for many applications a yes/no answer is not enough, and solvers are expected to generate either a satisfying assignment or a proof of unsatisfiability. For instance, proofs are useful for extracting interpolants (see [8] for a survey) to refine abstractions [20] or generate invariants [10].

CDCL-based SAT solvers generate proofs by propositional resolution [39]. Reasoners based on the DPLL(Σ) paradigm [2, 9, 26, 32] generate proofs by propositional resolution with proofs of theory lemmas plugged in as external subproofs (see [8], Section 2.4). This structure reflects a hierarchy where propositional logic is at the center and the theories are satellites. In DPLL(Σ) only propositional reasoning is conflict-driven, and non-conflict-driven theory solvers are integrated as black-boxes, so that their proofs also are black-boxes. In CDSAT, propositional logic and the SAT-solver lose their centrality as the sole conflict-driven component. Although propositional logic retains a special role, as all propagated assignments are Boolean, propositional logic is regarded as one of the theories, and all theory modules contribute directly to the proof, including new terms that may appear on the trail whence in the proof. For all these reasons, CDSAT proofs do not fit in the known mold of DPLL(Σ) proofs, and their generation requires machinery that goes beyond propositional resolution plus theory lemmas.

We present two approaches to proof generation in CDSAT. The first one is a proof-carrying CDSAT transition system (Section 4), where proof terms record the information needed to generate proofs. We describe different ways to turn proof terms into proofs (Section 5), including producing resolution proofs with theory lemmas. The proof objects produced by proof-carrying CDSAT can be directly checked by a verified checker [34] or exported to a proof format that can be verified by proof checkers. Thus, proof-carrying CDSAT can slot into pipelines from proof-search to proof-checking [1, 3, 4], where a minimal amount of proof information (e.g., an unsatisfiable core) might be sufficient for a theorem prover to regenerate a proof in its own format. The second approach consists of specifying a small kernel of primitives in LCF style [17, 30], as already explored in the Psyche prover [18], so that building proof objects in memory can be avoided. If CDSAT is implemented on top of this kernel, the LCF type abstraction ensures that an unsat answer is correct by construction, and CDSAT can be used as a trusted external oracle for interactive proof tools.

2 Background on CDSAT

CDSAT is an engine to determine the satisfiability of a quantifier-free formula modulo a combination of theories (SMT) and possibly modulo an initial assignment of values to some first-order variables or terms that appear in the formula. We call this generalization of SMT satisfiability modulo assignment (SMA). CDSAT combines in a conflict-driven manner inference systems, called theory modules, for component theories T1, . . . , Tn. The component theories are required to be disjoint so that their signatures Σ1, . . . , Σn share neither predicate nor function symbols, except equality, which is present in all signatures and for all sorts. In addition to equality, the theories typically share sorts and constant symbols.

Let T∞ be the union of T1, . . . , Tn. If Tk=(Sk, Fk), where Sk is the set of sorts and Fk is the set of symbols, for all k, 1 ≤ k ≤ n, the signature of T∞ is given by Σ∞=(Sk, Fk), with Sk=∪k≤n Sk and Fk=∪k≤n Fk. For most problems, one of the component theories is propositional logic, or the theory of Booleans, with logical connectives such as ∧, ∨, and ¬ as function symbols. For deciding the satisfiability of a set of literals in one or more convex theories (see [8] Definition 14, from [31] page 252), propositional logic is not needed as a component theory. If there is a non-convex theory (e.g., arrays), disjunction and therefore propositional reasoning are needed. Regardless of whether propositional logic is included, all theories T1, . . . , Tn must be able to interpret the truth values true and false and therefore their signatures are required to have the sort prop = {true, false}, where prop comes from proposition. Then, formulae are terms of sort prop. We use I for formulæ and i and u for terms of any sort.
CDSAT and theory modules manipulate assignments. An input problem \( \{l_1, \ldots, l_m\} \), where \( l_1, \ldots, l_m \) are formulæ, is seen as an initial assignment \( \{l_1 \leftarrow \text{true}, \ldots, l_m \leftarrow \text{true}\} \), even when abbreviated as \( \{l_1, \ldots, l_m\} \). Assignments, including initial assignments, are not restricted to assigning truth values; they can assign values of any sort. For example, with propositional logic, a fragment of arithmetic, and the theory of arrays, \( (x > 1) \leftarrow \text{false}, (x > 1) \wedge (y < 0) \leftarrow \text{true}, y \leftarrow -1, z \leftarrow \sqrt{2}, \) \( \text{store}(a, i, v) \equiv \text{false}, \text{select}(a, j) \equiv -3, \) and \( \text{select}(a, j) \equiv v \leftarrow \text{true} \) are all assignments. An assignment that assigns only values of sort prop is called a first-order assignment, as it assigns a value to a first-order term. If the input problem is an SMT problem, it is given to CDSAT as a Boolean assignment; if it is an SMA problem, it may also include first-order assignments.

First-order assignments are a standard ingredient in the definition of a first-order model. However, in automated reasoning, models need to be represented syntactically. First-order assignments raise the issue of what is a value that can be assigned. The notion of using ground terms to represent assignments that are neither due to theory inferences nor assignments that are neither due to theory inferences nor initial assignments will be placed on the trail by conflict-solving transitions. Every assignment on the trail has an empty justification, so that a derivation for the input problem \( \Gamma \) starts with the trail \( A \). A trail can be used as an assignment by ignoring order and justifications. The notion of endorsement given here corresponds to that of view-endorsement (see [7], Definition 7), obtained by combining the notions of endorsement for a single theory (see [7], Definition 6) and theory view of an assignment (see [7], Definition 2), omitted here for reasons of space.

Definition 2.2 (Level). Given a trail \( \Gamma \) with assignments \( A_0, \ldots, A_m \), the level of a singleton assignment \( A_i, 0 \leq i \leq m \), is given by level\(_L\)(\( A_i \)) = 1 + \max\{level\(_L\)(A) | j < i \}, if \( A_i \) is a decision, and level\(_L\)(\( A_i \)) = level\(_L\)(H), if \( A_i \) is a justified assignment with justification \( H \). The level of a set of singleton assignments \( H \subseteq \Gamma \) is given by level\(_L\)(H) = 0, if \( H = 0 \), and level\(_L\)(H) = \max\{level\(_L\)(A) | A \in H \}, otherwise.

\( \text{Lemma 2.3 (Level).} \)
An assignment of level $m$ may be added after one of level $r$, $r > m$; $\Gamma^{\leq m}$ denotes the restriction of $\Gamma$ to its elements of level at most $m$. The rules of the CDSAT transition system comprise search rules, whose application is denoted by $\rightarrow$, and conflict resolution rules, whose application is denoted by $\Rightarrow$, with transitive closure $\Rightarrow^*$ (see Fig. 1). The CDSAT rules may place on the trail assignments for new terms, meaning terms that do not appear in the input. For termination, these terms must come from a finite set $\mathcal{B}$ of terms, called global basis (cf. [7], Section 6.2). $\mathcal{B}$ is a parameter of the CDSAT transition system, whose instantiation depends on the input problem. The distinction between terms and $\mathcal{T}^+$-values plays a role here, as terms come from $\mathcal{B}$ and $\mathcal{T}^+$-values come from $\mathcal{P}_\infty$: $\mathcal{B}$ is finite and fixed once the input is given and throughout the derivation, whereas $\mathcal{P}_\infty$ may be an infinite supply out of which a derivation uses a finite subset that is not fixed prior to the start of the derivation. An assignment $H$ is in $\mathcal{B}$ if $t \in \mathcal{B}$ for all $t$ such that $(t \leftarrow c) \in H$.

Rule Decide adds to the trail $\Gamma$ a singleton assignment that is a decision, provided the term is relevant for a theory and the assignment acceptable for the theory module. Relevance (see [7], Definition 4) organizes the division of labor among theories. Acceptability (see [7], Definition 3) prevents adding an assignment that is already in $\Gamma$ or that is so immediately inconsistent with $\Gamma$ that it would have to be withdrawn without learning anything, such as adding $L$ if $\vec{L}$ is in $\Gamma$ or $1 \leftarrow 3$ in arithmetic with 1 a constant symbol in $\mathcal{P}_\infty$. Acceptability and relevance also imply that $A$ is in $\mathcal{B}$, which is relevant for termination [6, 7]. Rule Deduce extends $\Gamma$ with a Boolean singleton assignment justified by a theory inference $J \Gamma_k L$ from assignments $J$ already in $\Gamma$. The system proceeds with decisions and deductions until a conflict arises, as $J \Gamma_k L$, for some $k$, $1 \leq k \leq n$, and $\vec{L} \in \Gamma$: the assignment $J \cup \{\vec{L}\}$ is a conflict, because it is unsatisfiable. If the conflict is at level 0, rule Fail reports unsatisfiability. If the conflict is at a level greater than 0, rule ConflictSolve passes the control to the conflict resolution rules, and resumes the search from the trail $\Gamma'$ resulting from solving the conflict.

The conflict resolution rules operate on a trail and a conflict that is an unsatisfiable set of assignments from the trail. These rules transform the conflict until either it is solved or it surfaces at level 0 so that Fail fires. The rules use the notation $\cup$ for disjoint set union. Rule Undo applies if the conflict includes a first-order decision $A$ whose level $m$ is greater than that of any other element in the conflict: the effect of the rule is to exit the conflict by jumping back to level $m - 1$, removing $A$ and all its consequences from the trail. While it may seem that the resulting trail was already encountered, jeopardising termination, this is not the case [6]. Acceptability ensures that $A$ did not cause an inconsistency right away when it was added to the trail. If $A$ is now part of a conflict, it means that some justified assignment $L$ was added to the trail after decision $A$ even if $\text{level}_T(L) < m$ (late propagation). Thus, $\Gamma^{\leq m - 1}$ is new because it contains $L$.

Rule Resolve explains the conflict, by replacing a justified assignment $A$ with its justification $H$, unless $H$ contains a first-order decision $A'$ whose level $m$ is that of the current conflict. In that case, using Resolve to replace $A$ by its justification $H$ is forbidden, and it can be shown that $A$ must be a Boolean assignment $L$ also of level $m$ so that another

<table>
<thead>
<tr>
<th>SEARCH RULES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Decide $\Gamma \rightarrow \Gamma, \gamma A$</td>
<td>if $A$ is a $\mathcal{T}_L$-assignment for a term that is $\mathcal{T}_L$-relevant for $\Gamma$, and $A$ is acceptable for the $\mathcal{T}_L$-module and its view of $\Gamma$, with $1 \leq k \leq n$.</td>
</tr>
<tr>
<td>The next three rules share the conditions: $J \subseteq \Gamma$, $(J \Gamma_k L)$, and $L \notin \Gamma$, for some $k$, $1 \leq k \leq n$.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONFLICT RESOLUTION RULES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Undo $(\Gamma; E \cup {A}) \Rightarrow \Gamma^{\leq m - 1}$</td>
<td>if $A$ is a first-order decision of level $m &gt; \text{level}_T(E)$.</td>
</tr>
<tr>
<td>Resolve $(\Gamma; E \cup {A}) \Rightarrow (\Gamma; E \cup H)$</td>
<td>if $H_t A$ is in $\Gamma$ and $H$ does not contain a first-order decision $A'$ whose level is $\text{level}_T(E \cup {A})$.</td>
</tr>
<tr>
<td>UndoDecide $(\Gamma; E \cup {L}) \Rightarrow \Gamma^{\leq m - 1}, \vec{L}$</td>
<td>if $H_t L$ is in $\Gamma$, $m = \text{level}_T(E) = \text{level}_T(L)$ and $H$ contains a first-order decision $A'$ of level $m$.</td>
</tr>
<tr>
<td>LearnBackjump $(\Gamma; E \cup H) \Rightarrow \Gamma^{\leq m}, \vec{E} \cup L$</td>
<td>if $L$ is a clausal form of $H$ and is in $\mathcal{B}$, $L \notin \Gamma, \vec{L} \notin \Gamma$, and $\text{level}_T(E) \leq m &lt; \text{level}_T(H)$.</td>
</tr>
</tbody>
</table>

Figure 1. The CDSAT transition system with lemma learning.
rule applies \([6]\). Indeed, if an assignment other than \(L\) in the conflict has level \(m\), rule UndoDecide undoes \(A'\) and replaces it by a Boolean decision on \(\overline{L}\). If only one of the assignments in the conflict has the same level as the conflict, then the conflict is solved by a Backjump rule \([6, 7]\), replaced here with LearnBackjump to add learning as illustrated in Section 3.

The CDSAT transition system is non-deterministic as the rules leave room for heuristic choices. Thus, there are multiple CDSAT-derivations from a given input problem. In order to get a CDSAT procedure that yields a unique CDSAT-derivation from a given input, the transition rules must be coupled with a search plan that drives their application.

3 CDSAT with Lemma Learning

In order to enrich the CDSAT transition system with lemma learning, we turn assignments into clauses that the system can learn. First, we do this only for assignments that partake in conflicts, because in a conflict-driven system learning is also conflict-driven \([27]\). Second, only Boolean assignments can be turned into clauses. Third, in order to write non-unit clauses, we need disjunction. If propositional logic is not one of the combined theories, disjunction is not in the combined signature, and the only assignments that can yield clauses are singleton Boolean assignments that yield unit clauses. If propositional logic is one of the combined theories, any Boolean assignment can yield a clause.

Suppose that \(E \not\vdash H\) is a conflict, where \(H\) contains only Boolean assignments. This means that \(E \not\vdash H \models \bot\). If \(H\) is a singleton \(L\), we have \(E \not\vdash \{L\} \models \bot\), hence \(E \models \overline{L}\), and the flip \(\overline{L}\) is the clause associated to assignment \(L\) in the conflict. If \(H\) is not a singleton, it can be rewritten as the singleton assignment

\[
((\bigwedge (l \models \text{true} \in H) ) \land (\bigwedge (l \models \text{false} \in H \models \bot)) ) \models \text{true}
\]

can be flipped into

\[
((\bigwedge (l \models \text{true} \in H) ) \land (\bigwedge (l \models \text{false} \in H \models \bot)) ) \models \text{false}
\]

In order to get a clause, the above assignment can be rewritten in the equivalent form

\[
((\bigvee (l \models \text{true} \in H \models \bot)) \lor (\bigvee (l \models \text{false} \in H \models \bot)) ) \models \text{true}
\]

leading to the next definition.

**Definition 3.1** (Clausal form of an assignment in a conflict). Given a conflict \(E \not\vdash H\), where \(H\) is a Boolean assignment, the singleton Boolean assignments

\[
((\bigvee (l \models \text{true} \in H \models \bot)) \lor (\bigvee (l \models \text{false} \in H \models \bot)) ) \models \text{true}
\]

\[
((\bigwedge (l \models \text{true} \in H) ) \land (\bigwedge (l \models \text{false} \in H \models \bot)) ) \models \text{false}
\]

are clausal forms of \(H\).

We now describe the new rule LearnBackjump, listed last in Fig. 1. Rule LearnBackjump is quite versatile, since

1. It allows CDSAT to perform learning and backjumping;
2. It allows CDSAT to perform learning and restart;
3. It covers as a special case the Backjump rule of the basic version of CDSAT \([7]\), adding the capability to learn assertion clauses.

Input problem \(H_0\) including:

\[((-l_2 \lor l_5), (-l_2 \lor -l_2 \lor l_5))\]

Initial trail \(\Gamma_0\) including:

\[\emptyset, (-l_2 \lor l_5), (-l_2 \lor -l_2 \lor l_5)\]

Extending \(\Gamma_0\) into \(\Gamma = \Gamma_0 \cup \{l_2, l_3, l_4, \ldots, l_5\}\)

(involving unrelated decisions \(A_1\) and \(A_2\))

First conflict:

\[(\Gamma; (-l_2 \lor -l_2 \lor -l_2 \lor l_5))\]

Applying Resolve to \(l_3:\) \(\langle l_3; (-l_2 \lor -l_2 \lor -l_2 \lor l_5), l_2, l_4, (-l_2 \lor -l_2 \lor l_5)\rangle\]

**Figure 2.** Propositional extract from a CDSAT derivation

We consider these three features in this order.

**Learning and backjumping** is the generic behavior of rule LearnBackjump. This rule singles out a Boolean subset \(H\) of the conflict \(E \not\vdash H\), such that \(\text{level}(E) > \text{level}(H)\). Then, it solves the conflict by jumping back to a level \(m\), such that \(\text{level}(E) \leq m < \text{level}(H)\), and learning a clausal form \(L\) of \(H\). The system learns \(L\) by adding to the trail the justified assignment \(E \models L\), since \(E \not\vdash H \models \bot\) implies \(E \models \overline{L}\), as \(L\) is a clausal form of \(H\). The clausal form \(L\) may assign a Boolean value to a new term, and therefore it must be an element of \(B\). Note that \(H\) does not necessarily contain all the Boolean assignments in the conflict: the choices of the Boolean subset \(H\) and the destination level \(m\) are left to the search plan.

**Example 3.2.** Consider the conflict on the last line of Fig. 2. Assume that LearnBackjump is applied with \(H = \{l_2, l_4\}\), \(E = \{(-l_2 \lor -l_5 \lor -l_2), (-l_2 \lor -l_5)\}\), where \(\{(-l_2 \lor -l_5) \models \text{true}\) is a clausal form of \(H\). We have \(\text{level}(E) = 4\) and \(\text{level}(L) = 0\), so that any destination level \(m\) such that \(0 < m < 4\) can be picked. A standard choice for \(m\) would be the second highest level in the conflict, namely \(m = 2\), in which case the LearnBackjump step jumps over decision \(A_3\) and yields

\[\Gamma_0 \cup A_1, \{l_2 \models l_2 \lor l_3 \lor l_4 \lor l_5\}

The derivation continues from level \(2\) with the learned lemma \(\overline{l_2} \lor -l_4\) added to level \(0\).

We consider next learning and restart. It is common to restart after learning a clause, and search plans with aggressive restart proved successful in SAT solving. LearnBackjump makes this kind of search plan possible in CDSAT. Assume that the destination level \(m\) is chosen to be the smallest, that is, \(m = \text{level}(E)\). If \(\text{level}(E) = 0\), the shape of the trail after LearnBackjump is \(\Gamma \models \emptyset, E, L\), which means LearnBackjump performs a restart and adds \(E\) to level \(0\).

**Example 3.3.** Consider the same LearnBackjump step as in Example 3.2 except that the destination level is \(m = 0\). We get

\[\Gamma_0, (-l_2 \lor -l_2 \lor -l_2), (-l_2 \lor -l_2 \lor l_5)\]

as all decisions are undone and the derivation restarts with the learned lemma \(\overline{l_2} \lor -l_4\) added to level \(0\).

In the rest of this section we analyze how LearnBackjump subsumes Backjump \([7]\) and allows CDSAT to learn assertion clauses. The Backjump rule applies when CDSAT reaches a conflict state \(\langle \Gamma; E \not\vdash \{L\} \rangle\), where \(\text{level}(L) > m\), with
m = level(E). Backjump solves such a conflict by producing \( \Gamma \leq m \), \( \Gamma \). In words, it jumps back to level \( m \) and adds to the trail the justified assignment \( \Gamma \), because \( E \cup \{ l \} \models \perp \) yields \( E \models \perp \). Rule LearnBackjump can behave in the same way by taking as \( H \) a singleton \( L \). Indeed, \( \perp \) is a clausal form of any singleton Boolean assignment \( L \) in the conflict. However, even when it reproduces Backjump, LearnBackjump allows a choice of destination level: Backjump goes back to level \( m \) = level(E), while LearnBackjump allows the system to go back to any level \( m \) such that level(E) \( \leq m < \) level(E).

**Example 3.4.** The conflict on the last line of Fig. 2 contains an assignment, namely \( l_4 \), whose level is greater than that of the rest of the assignment, as level\( (l_4) = 4 > \) level\( (E) = 2 \) where \( E = \{ (\neg l_2 \vee l_4 \neg l_4), l_2, (\neg l_4 \vee l_3) \} \). Thus, Backjump could apply and we can see how LearnBackjump mimics it. A LearnBackjump step with \( H = \{ l_4 \} \) and \( m = 2 \) yields

\[
\Gamma_0, \gamma A_1, \gamma l_2, \gamma \{ \neg l_2 \neg l_4 l_4 \}, l_2, (\neg l_4 \vee l_3) \triangleright \Gamma_4.
\]

Alternatively, if \( m = 3 \), LearnBackjump yields

\[
\Gamma_0, \gamma A_1, \gamma l_2, \gamma A_3, (\neg l_2 \neg l_4 l_4 \neg l_4) l_2, (\neg l_4 \vee l_3) \triangleright \Gamma_4.
\]

The side-condition \( L \notin \Gamma \) of LearnBackjump prevents adding to the trail clauses that are already there.

**Example 3.5.** LearnBackjump can be applied also to the first conflict in Fig. 2, namely \( \Gamma = \{ (\neg l_2 \vee l_4 \neg l_4), l_2, l_4, l_5 \} \). Consider a step with \( E = \{ \neg l_2 \vee l_4 \neg l_4 \} \), so that level\( (H) = 4 \) and level\( (E) = 0 \). Regardless of the choice of destination level \( m, 0 < m < 4 \), a clausal form of \( H \) is redundant since clause \( \neg l_2 \vee l_4 \neg l_4 \) is already on the trail and LearnBackjump does not add it.

Unlike Backjump, LearnBackjump does not require the occurrence in the conflict of a singleton assignment \( L \) of level greater than the rest of the conflict.

**Example 3.6.** In the first conflict in Fig. 2 both \( l_4 \) and \( l_5 \) have level 4. If we apply LearnBackjump with \( H = \{ l_4 \}, E = \{ (\neg l_2 \vee l_4 \neg l_4), l_2 \} \), level\( (H) = 4 \), level\( (E) = 2 \), and destination level \( m = 2 \), the resulting trail is

\[
\Gamma_0, \gamma A_1, \gamma l_2, \gamma \{ \neg l_2 \neg l_4 \} \triangleright \Gamma_4
\]

where \( \neg l_2 \vee l_4 \neg l_4 \) \( \models \) true on level 2 is a clausal form of \( H \).

The last conflict clause generated prior to backjumping is called *backjump clause*: the system learns this clause and jumps back to a prior level in such a way to undo at least one decision and satisfy the learned clause by placing on the trail one of its literals. The First Unique Implication Point (1UIP) heuristic [28] picks as the backjump clause the first generated assertion clause, where an *assertion clause* is a conflict clause where only one of its literals, say \( l \), is falsified at the current level. The Backjump rule of CDSAT generalizes this principle, because it applies when the conflict contains a singleton assignment \( L \) of level greater than the rest of the conflict (not necessarily the current one). However, Backjump places \( \perp \) on the trail without learning an assertion clause. When a CDSAT conflict contains a singleton assignment \( L \) of level greater than the rest of the conflict, it means it is possible to extract from the conflict an assertion clause. Thus, with rule LearnBackjump, CDSAT can learn assertion clauses, and a CDSAT search plan may choose to restrict the application of this rule to 1UIP conflict clauses.

Let \( k = l_1 \vee \cdots \vee l_q \) be an assertion clause that is in the global basis \( \mathcal{B} \). Let \( l_q \) be the literal of \( k \) made false by the current level and \( L \) the assignment that makes \( l_q \) false. In order to learn \( k \), it suffices to identify the Boolean part \( H = H' \cup \{ L \} \) of the conflict that makes \( k \) false: for all \( i, 1 \leq i \leq q, (l_i \leftarrow \text{true}) \in H \) if and only if \( l_i \in \mathcal{L} \) and \( l_i \leftarrow \text{false} \) if in \( H' \) if and only if \( l_i \in \mathcal{K} \). By Definition 3.1, the assignment \( K = (k \leftarrow \text{true}) \) is a clausal form of \( H' \). Let \( E \) be the rest of the conflict. Then the system applies LearnBackjump with destination level \( m = \) level\( (E \cup \{ H' \}) \), which means \( m = \) level\( (E) > \) level\( (H' \}) \), \( m = \) level\( (H') \) \( < \) level\( (E) \), and \( m = \) level\( (H') \) = level\( (E) \) if level\( (H') \) = level\( (E) \). This choice satisfies the condition level\( (E) \leq m < \) level\( (H) \), because level\( (E) \leq \) level\( (E \cup H') < \) level\( (H) \) = level\( (E) \). This LearnBackjump step yields the trail

\[
\Gamma \leq m, \Gamma \leq \perp, \Gamma \leq E_{\mathcal{K}}, \Gamma \leq H' \triangleright
\]

and \( k \) is learned. The theory module for propositional logic features inference rules for unit propagation (cf. [7], Section 4.1) that allow the inference:

\[
\{ K \} \models H' \triangleright \perp
\]

(1)

Indeed, \( K \) is \( (l_1 \vee \cdots \vee l_q \) \( \models \) true, and \( H' \) makes \( l_1, \ldots, l_{q-1} \) false, so that unit propagation infers \( l_q \). Since \( L \) makes \( l_q \) false, \( L \) makes \( l_q \) true. Because the destination level \( m \) of the LearnBackjump step was chosen in such a way that \( m > \) level\( (H') \), the premises \( K, H' \) of inference (1) are all on the trail \( \Gamma \leq m, \Gamma \leq \perp \). Furthermore, literal \( l_q \) is in \( \mathcal{B} \), since \( L \) was on the trail. Thus, all conditions are met to apply a Deduce step with inference (1). The resulting trail is

\[
\Gamma \leq m, \Gamma \leq \perp, \{ K \} \models H' \triangleright \perp
\]

which is similar to \( \Gamma \leq m, \Gamma \leq \perp, \mathcal{B} \), produced by Backjump, except for the learned clause \( K \). The advantage is that \( K \) can be reused in future branches of the search. The level of \( \Gamma \leq \perp \) is level\( (E) \); the smaller level\( (E) \), the longer \( K \) may remain on the trail and be used for inferences.

**Example 3.7.** Continuing Example 3.2 from

\[
\Gamma_0, \gamma A_1, \gamma l_2, \gamma \{ \neg l_2 \neg l_4 l_4 \neg l_4 \}, \gamma \{ \neg l_2 \neg l_4 \} \triangleright \Gamma_4
\]

rule Deduce with inference (1) generates

\[
\Gamma_0, \gamma A_1, \gamma l_2, \gamma \{ l_2 l_4 l_4 l_4 \}, \gamma \{ \neg l_2 \neg l_4 \} \triangleright \Gamma_4
\]

CDSAT with lemma learning inherits the soundness, completeness, and termination properties of CDSAT, under suitable assumptions on theory modules and the input-dependent...
global basis $\mathcal{B}$ (see [7], Section 6, and [6], Sections 8 and 9). Intuitively, $\mathcal{B}$ needs to be finite for termination, and sufficiently large for completeness. Informally speaking, the latter means that $\mathcal{B}$ contains all input terms and all terms that the theory modules may need, as defined by $n$ local basis for the combined theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$. Such a global basis is called stable. The theory modules need to be complete, which means that when none of them can extend the trail, the assignment on the trail is endorsed by a $\mathcal{T}_n^*$-model.

**Theorem 3.8.** CDSAT with lemma learning is sound (if it derives unsat, the input problem is unsatisfiable), complete (if the theory modules are complete, and the global basis is stable, whenever the derivation reaches a state other than unsat such that no transition rule applies, there exists a model that endorses the input problem), and terminating (every derivation with a finite global basis is guaranteed to terminate).

We leave for future work the performance analysis of different ways to apply LearnBackjump, focusing next on proof generation.

# 4 Proof Object Production in CDSAT

In this section we describe the first technique to endow CDSAT with **proof generation**. This technique enables the system to build proof objects in memory. When a CDSAT derivation terminates by firing rule Fail and detecting unsatisfiability, the system engages in an activity called **proof reconstruction** [5], which consists of extracting a proof object from the final state of the derivation. For this, we need to instrument the transition system in such a way that the final state contains enough information to reconstruct a proof object.

## 4.1 Theory-Specific Proofs

Because CDSAT combines theory modules, proof object production in CDSAT requires proof object production in every theory module. Therefore, we assume that each theory module is equipped with a **proof annotation system** that annotates theory inferences with **theory proofs**:

$$ J \vdash_{k} j_{k} : L $$

means that the module for theory $\mathcal{T}_k$ infers $L$ from $J$ with theory proof $j_k$. A theory proof refers to $L$ and the individual assignments in $J$ by means of **identifiers**: in $J \vdash_{k} j_{k} : L$, the premise $J$ is a $\mathcal{T}_k$-assignment with identifiers, that is, a set of triples $a(t \leftarrow c)$, where $a$ is the identifier of the singleton $\mathcal{T}_k$-assignment $t \leftarrow c$.

For instance, inference (1) from Section 3, representing Unit Propagation (UP) in the theory module for propositional logic, can be annotated with a theory proof denoted $UP(a, \{a_1, \ldots, a_n\})$; $L$,

$$ \{L \} \cup H \vdash_{\text{Boo}l} UP(a, \{a_1, \ldots, a_n\}); L $$

where $a_1, \ldots, a_n$ are the identifiers of the assignments in $H$. For Linear Rational Arithmetic (LRA) annotated inferences include instances of Fourier-Motzkin resolution

---

**Figure 3.** Equality inference rules for terms of sort $s$

\[
\begin{align*}
\alpha_1(t_1 \leftarrow c_1), \alpha_2(t_2 \leftarrow c_2) \vdash \text{eq}(a_1, a_2) : t_1 \equiv t_2 & \quad \text{if } c_1 \text{ and } c_2 \text{ are the same } \mathcal{T}^*\text{-value of sort } s \\
\alpha_1(t_1 \leftarrow c_1), \alpha_2(t_2 \leftarrow c_2) \vdash \text{neq}(a_1, a_2) : t_1 \neq t_2 & \quad \text{if } c_1 \text{ and } c_2 \text{ are distinct } \mathcal{T}^*\text{-values of sort } s \\
\alpha_1(t_1 \leftarrow c_1), \alpha_2(t_2 \leftarrow c_2) \vdash \text{sym}(a_1, a_2) : t_1 \equiv t_2 & \quad \text{if } c_1 \equiv c_2 \\
\end{align*}
\]

---

**Example 4.1.** If we have

\[
\begin{align*}
\alpha_1(t_1 \leftarrow c_1), \alpha_2(t_2 \leftarrow c_2) & \vdash \text{eq}(1, 4), f(x) \equiv f(y), \\
\text{we also have } & \quad \alpha_1(t_1 \leftarrow c_1), \alpha_2(t_2 \leftarrow c_2) \vdash \text{eq}(4, 1) : f(x) \equiv f(y). \\
\end{align*}
\]

---

**4.2 Proof Terms**

In order to empower CDSAT to generate proof objects, we instrument the CDSAT transition system to keep track of provability invariants. We call the outcome **proof-carrying CDSAT**. The provability invariants of a transition system ensure that the transitions do not change the problem. If
The provability invariants of CDSAT are:

1. For every assignment \( H_c A \) on the trail, \( H \models A \).
2. For every conflict state \( (\Gamma; E) \), \( E \models \bot \).

Proof-carrying CDSAT keeps track of provability invariants by means of a trace of proof terms. We distinguish between deduction proof terms and conflict proof terms, intuitively related to Invariants (1) and (2), respectively. A deduction proof term records why an assignment \( H_c A \) is on the trail, while a conflict proof term records why the second component of a conflict state \((\Gamma; E)\) is a conflict.

**Definition 4.2 (CDSAT proof terms).** A CDSAT proof term is either a deduction proof term

\[ j := \text{in}(A) | j_k | \text{lem}(H.c) \]

or a conflict proof term

\[ c := \text{cfl}(j_k, a) \mid \text{res}(j, {A.c}) \]

where in, lem, cfI, and res are the CDSAT proof constructors. \( j_k \) ranges over theory proofs for \( T_\Gamma \), \( 1 \leq k \leq n \), \( A \) is a singleton assignment, and \( H \) is a Boolean assignment with identifiers; \( \text{res}(j, {A.c}) \) binds \( a \) in \( c \), and \( \text{lem}(H.c) \) binds the identifiers of \( H \) in \( c \), as represented by the dot.

The proof system of Fig. 4 helps understanding the invariants that proof terms keep track of. The first three rules establish the derivability of judgments of the form \( H \vdash j : A \), witnessing invariant \( H \models A \). Proof term \( \text{in}(A) \), where in stands for initial, witnesses the fact that an initial assignment \( A \) holds. The second rule shows that a theory proof \( j_k \) can be coerced into a CDSAT deduction proof term. The rule for \( \text{lem}(H.c) \), where lem stands for lemma, shows that, whenever there is a conflict involving a Boolean assignment \( H \), a clausal form of \( H \) is entailed by the rest of the conflict and therefore it is a lemma. The proof term carries \( H \) to indicate which part of the conflict is turned into a lemma. Since identifiers of assignments in \( H \) may occur in \( c \), such occurrences are bound in \( \text{lem}(H.c) \).

The last two rules establish the derivability of judgments of the form \( E \vdash c : \bot \), witnessing invariant \( E \models \bot \). Proof term \( \text{cfl}(j_k, a) \), where \( \text{cfl} \) stands for conflict, witnesses the conflict between the conclusion \( L \) of a theory inference and its flip \( \overline{L} \) (having identifier \( a \)). Proof term \( \text{res}(j, {A.c}) \), where \( \text{res} \) stands for resolve, is the only construct that combines two subproofs, connecting the conclusion \( A \) of the left premise with the hypothesis \( {A.c} \) of the right premise. Any occurrence of \( a \) in \( c \) is bound in \( \text{res}(j, {A.c}) \). Using the soundness of theory inferences, a structural induction on proof terms turns the intuition of witnessing invariants into the following theorem.

**Theorem 4.3.** If \( H \vdash j : A \), then \( H \models A \); if \( E \vdash c : \bot \), then \( E \models \bot \).

This choice of proof terms and proof system allows us to associate to every rule of the CDSAT transition system the simplest trace of how the rule transforms the relevant provability invariant, leading us to the next section.

### 4.3 Proof-Carrying CDSAT

The proof-carrying CDSAT transition system is given in Fig. 5. While a decision \( \neg A \) does not carry a proof term, a justified assignment \( H_{\neg A} A \) carries a deduction proof term \( j \) such that \( H \vdash j : A \). Initial assignments take the form \( \text{in}(A) | A \) where the deduction proof term presents the initial assignment as a premise of the proof. If a theory inference \( j_k L \), \( 1 \leq k \leq n \), supports the justified assignment \( j \), \( L \), the same inference \( j_k L \) annotated with a theory proof supports the justified assignment \( j \). (see rule Deduce in Fig. 5).

Because proof terms may use identifiers of assignments, proof-carrying CDSAT manipulates assignments with identifiers, and the subset relation takes identifiers into account. In Fig. 5, the condition \( J \subseteq \Gamma \) means that \( J \) is a \( T_\Gamma \)-assignment with identifiers, and for every \( t \in \{ \top, \bot \} \), the pair \( t \vdash c \) appears in \( \Gamma \) with identifier \( a \). Conflict \( E \) is an assignment with identifiers such that \( E \vdash \bot \) and \( E \subseteq \Gamma \) with the same meaning. Conflict resolution rules operate on states of the form \( (\Gamma; E ; c) \), where \( c \) is a conflict proof term. An analysis of the rules shows that the invariants in Theorem 4.4 hold.

**Theorem 4.4.** For all proof-carrying CDSAT derivations

- If a trail containing \( H_{\neg A} A \) is generated, then \( H \vdash j : A \);
- If a conflict state \( (\Gamma; E ; c) \) is reached, then \( E \vdash c : \bot \).

A difference between the system in Fig. 5 and that in Fig. 1 is that in proof-carrying CDSAT the dichotomy between Fail and ConflictSolve is based on the outcome of the conflict resolution phase rather than the level of the conflict. Given Theorems 4.3 and 4.4, reaching state \( (\Gamma; \emptyset ; c) \) means that the input problem is unsatisfiable in \( T_\Gamma^c \) and \( c \) is a proof term witnessing its unsatisfiability. If such a state is reached, Fail fires and terminates the derivation in state unsat(\( c \)). It is simple to show that the conflict resolution rules in Fig. 5 reduce \( (\Gamma; E ; c_1) \) to a state of the form \( (\Gamma; \emptyset ; c) \) if and only if level(\( E \)) = 0; and that they solve the conflict producing some trail \( \Gamma' \) different from \( \Gamma \) if and only if level(\( E \)) > 0.

The CDSAT transition system of Fig. 1 returns unsat as soon as a conflict at level 0 is found, because it does not generate a proof. For proof-carrying CDSAT, detecting a conflict at level 0 does not suffice: in order to generate a proof of unsatisfiability of the initial assignment, proof-carrying CDSAT...
integrates a proof of why \( E \) follows from the initial assignment with a proof of why \( E \) is unsatisfiable. This is achieved by proof-carrying \( \text{Resolve} \), which combines proof term \( c \), witnessing the unsatisfiability of the conflict, with proof term \( j \) witnessing that one of the assignments in the conflict, named \( A \), follows from prior assignments: \( A \) is retained in the proof term \( \text{res}(j, A_{\text{c}}) \). Proof-carrying \( \text{LearnBackjump} \) generates the proof term \( \text{lem}(H, c) \), recording that the learned lemma \( L \) is a clause form of \( H \), and turning the conflict proof term \( c \) representing a proof of unsatisfiability of \( E \oplus H \) into a deduction proof term representing a proof of \( L \) from \( E \).

### 4.4 Example of Proof-Carrying Derivation

We now illustrate some operations of proof-carrying CDSAT. Fig. 6 gives a refutation for the full version of the example in Fig. 2. Adding identifiers to the first conflict in Fig. 2 we get

\[
\begin{align*}
\text{c}_1 &= \text{cfl}(\text{UP}(a_1, \{a_2, a_4\}, a_5)) \\
\text{c}_2 &= \text{res}(\text{UP}(a_6, \{a_4\}), d_1, \text{c}_1)
\end{align*}
\]

The proof term for this conflict involves Unit Propagation:

\[
\begin{align*}
\text{c}_1 &= \text{cfl}(\text{UP}(a_7, \{a_2, a_4\}, a_5))
\end{align*}
\]

The next step replaces \( d_1 \) in the conflict by its justification \( a_3 = (\lnot l_2 \lor \lnot l_3) \lor a_1 l_2, a_1 l_4, a_5 l_5 \).

The proof term for this conflict involves Unit Propagation:

\[
\begin{align*}
\text{c}_1 &= \text{cfl}(\text{UP}(a_7, \{a_2, a_4\}, a_5)) \\
\text{c}_2 &= \text{res}(\text{UP}(a_6, \{a_4\}), d_1, \text{c}_1)
\end{align*}
\]

where \( \text{UP}(a_6, \{a_4\}) \) is the theory proof that justifies why \( b_3 \) is on the trail as a result of Unit Propagation. The conflict is solved by \( \text{LearnBackjump} \) as in Example 3.2, learning \( \langle \lnot l_2 \lor \lnot l_3 \rangle \lor \langle \lnot l_4 \rangle \) on the trail (line 3), where \( j_1 = \text{lem}(\{a_1 l_2, a_1 l_4\}, c_2) \).

These patterns repeat several times in Fig. 6. For example, \( j_2, j_5, j_6, j_9, j_{10} \) (lines 4, 9, 10, 17, 18) are deduction proof terms of the form \( \text{UP}(\_\_\_), \) as they support unit propagations, and similarly to \( c_1 \), proof term \( c_4 \) for conflict \( \langle \lnot l_2 \lor \lnot l_3 \rangle \lor a_1 l_2, a_1 l_4 \) plugs in a unit propagation subproof (line 19):

\[
\begin{align*}
\text{c}_4 &= \text{cfl}(\text{UP}(a_7, \{a_2\}, a_4))
\end{align*}
\]

The derivation involves the LRA inferences (line 5)

\[
\begin{align*}
\text{c}_5 &= \text{cfl}(\text{FM}(a_2, a_3))
\end{align*}
\]

where \( j_3 = \text{eval}(\{a_1\}) \) and the LRA inferences (line 6)

\[
\begin{align*}
\text{c}_6 &= (y \geq 0), a_6(y+0), a_7(x \leq 0) \vdash c_3 \vdash \bot
\end{align*}
\]

where \( c_3 = \text{cfl}(\text{FM}(a_2, a_3), a_3) \) as

\[
\begin{align*}
\text{c}_7 &= (0 \leq y), a_7(y \leq x) \vdash \text{LRA} \text{FM}(a_2, a_3); (0 \leq x).
\end{align*}
\]

It also employs the EUF inferences (lines 11 and 12)

\[
\begin{align*}
\text{c}_8 &= (f(z) \rightarrow \text{blue}), a_8(f(y) \rightarrow \text{red}) \vdash j_7(f(z) \neq f(y)) \vdash c_5 \vdash \bot
\end{align*}
\]

where \( j_7 = \text{neq}(a_{10}, a_{11}) \) and \( c_5 = \text{cfl}(\text{Cong}(a_{12}), a_{13}) \). Deduction proof terms \( j_8 \) and \( j_9 \) (lines 8 and 16) justify the learning of lemmas \((x \leq 0) \lor \lnot l_2 \land (x \leq 0) \), respectively; therefore they instantiate the same pattern as \( j_1 \), namely \( \text{lem}(\_\_\_, \) \). The conflict proof terms \( c_{14}, c_{16}, c_{17}, \) and \( c_9 \) through \( c_{21} \) all encode \( \text{Resolve} \) steps, and therefore are of the form \( \text{res}(\_\_\_, \_\_), \) like \( c_2 \). More precisely, \( c_{14}, c_{15} = c_{18}, c_{20} \), and \( c_{21} \) are of the form \( \text{res}(\_\_\_, \_\_, \_\_), \) whose effect is simply to remove an assignment from the conflict because it is an initial assignment. In contrast, conflict proof term \( c_{19} \) is of the form

\[
\begin{align*}
\text{c}_{19} &= \text{cfl}(\text{UP}(a_7, \{a_2\}, a_{18}))
\end{align*}
\]

where \( a \) is the identifier of the learned clause \( \langle \lnot l_2 \lor \lnot l_3 \rangle \) line
Input problem: \((\neg l_4 \lor l_3), (\neg l_2 \lor \neg l_4 \lor \neg l_5), (l_2 \lor (z = y)), (\neg (x \leq 0) \lor l_2), (\neg (x \leq 0) \lor l_4), (f(z) \rightarrow \text{blue}), (f(y) \rightarrow \text{red})\)

where \(l_2\) is \((y \geq 0)\), and \(l_4\) is \((x+y>0)\). Assume decision \(A_1\) is \((x \rightarrow l_4)\). Detected conflicts are introduced with symbol \(\Delta\).

To improve readability, identifiers in justifications and conflicts are omitted; they are determined by the current trail.

\[
\Delta \quad (\langle \Gamma; (\neg l_2 \lor \neg l_2 \lor \neg l_5), l_2, l_4, l_5 \triangleright c_1 \rangle \quad \Rightarrow \quad (\langle \Gamma; (\neg l_2 \lor \neg l_2 \lor \neg l_5), l_2, l_4, (\neg l_4 \lor \neg l_2) \triangleright c_2 \rangle)
n \quad \Rightarrow \quad \Gamma_0, \gamma A_1, \forall l_2, \forall (\neg l_2 \lor \neg l_2 \lor \neg l_5), (\neg l_4 \lor \neg l_2) \triangleright \Gamma_0
\]

\[
\quad \Rightarrow \quad \Gamma_0, \gamma A_1, l_2, (\neg l_2 \lor \neg l_2 \lor \neg l_5), (\neg l_4 \lor \neg l_2) \triangleright j_1, (\neg l_2 \lor \neg l_4) \triangleright \Gamma_4
\]

\[
\quad \Rightarrow \quad \Gamma_0, \gamma A_1, l_2, (\neg l_2 \lor \neg l_2 \lor \neg l_5), (\neg l_4 \lor \neg l_2) \triangleright j_1, (\neg l_2 \lor \neg l_4) \triangleright \Gamma_5, A_1^{j_1+j_2}, (x \leq 0) =: \Gamma_1
\]

\[
\Delta \quad \langle \Gamma_1; l_2, \Gamma_4, (x \leq 0) \triangleright c_3 \rangle \quad \Rightarrow \quad (\langle \Gamma_1; l_2, (\neg l_2 \lor \neg l_4), (x \leq 0) \triangleright c_4 \rangle)
\]

\[
\quad \Rightarrow \quad \Gamma_0, \gamma A_1, (\neg l_2 \lor \neg l_2 \lor \neg l_5), (\neg l_4 \lor \neg l_2) \triangleright j_1, (\neg l_2 \lor \neg l_4) \triangleright \Gamma_5
\]

\[
\quad \Rightarrow \quad \Gamma_0, \gamma A_1, l_2, (\neg l_2 \lor \neg l_2 \lor \neg l_5), (\neg l_4 \lor \neg l_2) \triangleright j_1, (\neg l_2 \lor \neg l_4) \triangleright \Gamma_5, A_1^{j_1+j_2}, (x \leq 0) =: \Gamma_2
\]

\[
\Delta \quad \langle \Gamma_2; l_2, \Gamma_6, (x \leq 0) \triangleright c_5 \rangle \quad \Rightarrow \quad (\langle \Gamma_2; l_2, (\neg l_2 \lor \neg l_4), (x \leq 0) \triangleright c_6 \rangle)
\]

\[
\quad \Rightarrow \quad \Gamma_0, \gamma A_1, (\neg l_2 \lor \neg l_2 \lor \neg l_5), (\neg l_4 \lor \neg l_2) \triangleright j_1, (\neg l_2 \lor \neg l_4) \triangleright \Gamma_5
\]

\[
\quad \Rightarrow \quad \Gamma_0, \gamma A_1, l_2, (\neg l_2 \lor \neg l_2 \lor \neg l_5), (\neg l_4 \lor \neg l_2) \triangleright j_1, (\neg l_2 \lor \neg l_4) \triangleright \Gamma_5, A_1^{j_1+j_2}, (x \leq 0) =: \Gamma_3
\]

\[
\Delta \quad \langle \Gamma_3; (f(0) \neq f(0)), (f(0) \neq f(0)), (l_2 \lor (z = y)), l_2 \triangleright j_4 \rangle \quad \Rightarrow \quad (\langle \Gamma_3; (f(0) \neq f(0)), (f(0) \neq f(0)), (l_2 \lor (z = y)), l_2 \triangleright c_7 \rangle)
\]

\[
\quad \Rightarrow \quad \Gamma_0, \gamma A_1, (\neg l_2 \lor \neg l_2 \lor \neg l_5), (\neg l_4 \lor \neg l_2) \triangleright j_1, (\neg l_2 \lor \neg l_4), (\neg l_4 \lor \neg l_2) \triangleright j_1, (x \leq 0) \lor \neg l_2 =: \Gamma_4
\]

\[
\quad \Rightarrow \quad \Gamma_0, \gamma A_1, l_2, (\neg l_2 \lor \neg l_2 \lor \neg l_5), (\neg l_4 \lor \neg l_2) \triangleright j_1, (\neg l_2 \lor \neg l_4) \triangleright \Gamma_5
\]

\[
\quad \Rightarrow \quad \Gamma_0, \gamma A_1, l_2, (\neg l_2 \lor \neg l_2 \lor \neg l_5), (\neg l_4 \lor \neg l_2) \triangleright j_1, (\neg l_2 \lor \neg l_4) \triangleright \Gamma_5, A_1^{j_1+j_2}, (x \leq 0) =: \Gamma_5
\]

\[
\Delta \quad \langle \Gamma_5; (\neg l_2 \lor \neg l_4), l_4 \triangleright c_8 \rangle \quad \Rightarrow \quad (\langle \Gamma_5; (\neg l_2 \lor \neg l_4), l_4 \triangleright c_9 \rangle)
\]

\[
\quad \Rightarrow \quad \Gamma_5, (\neg l_2 \lor \neg l_4), l_4 \triangleright c_{10}
\]

\[
\quad \Rightarrow \quad \Gamma_5, (\neg l_2 \lor \neg l_4), l_4 \triangleright c_{11}
\]

\[
\quad \Rightarrow \quad \Gamma_5, (\neg l_2 \lor \neg l_4), l_4 \triangleright c_{12}
\]

\[
\quad \Rightarrow \quad \Gamma_5, (\neg l_2 \lor \neg l_4), l_4 \triangleright c_{13}
\]

\[
\quad \Rightarrow \quad \Gamma_5, (\neg l_2 \lor \neg l_4), l_4 \triangleright c_{14}
\]

\[
\quad \Rightarrow \quad \Gamma_5, (\neg l_2 \lor \neg l_4), l_4 \triangleright c_{15}
\]

\[
\quad \Rightarrow \quad \Gamma_5, (\neg l_2 \lor \neg l_4), l_4 \triangleright c_{16}
\]

\[
\quad \Rightarrow \quad \Gamma_5, (\neg l_2 \lor \neg l_4), l_4 \triangleright c_{17}
\]

\[
\quad \Rightarrow \quad \Gamma_5, (\neg l_2 \lor \neg l_4), l_4 \triangleright c_{18}
\]

\[
\quad \Rightarrow \quad \Gamma_5, (\neg l_2 \lor \neg l_4), l_4 \triangleright c_{19}
\]

\[
\quad \Rightarrow \quad \Gamma_5, (\neg l_2 \lor \neg l_4), l_4 \triangleright c_{20}
\]

\[
\quad \Rightarrow \quad \Gamma_5, \emptyset ; c_{21}
\]

\[
\Rightarrow \quad \text{unsat}(c_{21})
\]

\[
\text{Figure 6. A proof-carrying continuation of the CDSAT derivation in Fig. 2}
\]

30. Note that \(a\) occurs twice in \(c_{19}\); once in the subproof \(c_8\) (line 19), as seen above, and once in the subproof (line 23)

\[
c_{21} = \text{res}(j_4, \alpha l_4, (x \leq 0) \lor \neg l_2, c_{11}),
\]

since \(j_4 = \text{lcm}\{(\neg (x \leq 0), \alpha l_4), c_{11}\}\) (line 8), \(c_4 = \text{res}(j_2, \alpha l_4, c_3)\) (line 7), and \(j_2 = \text{UP}(a, \{a_2\})\) (line 4). This double occurrence reflects the fact that lemma \((\neg l_2 \lor \neg l_4)\) has been reused in the derivation, which is a purpose of learning. Had Backjump been applied instead of LearnBackjump (cf. Example 3.4), the trail would have been extended (line 3) with

\[
(\neg l_2 \lor \neg l_4, \neg l_4 \lor l_5, l_2 \lor j_1, \Gamma_4)
\]

where the deduction proof \(j_1 = \text{lcm}(\alpha l_4, c_2)\) justifies why \(l_4\) is on the trail. If this route had been followed, the conflict reflected by \(c_3\) (line 19) would not have been detected. More steps would have been necessary to discover it, amounting to constructing another proof of \((\neg l_2 \lor \neg l_4)\), possibly identical to the first one that was not recorded.

A full development of the final conflict proof term \(c_{21}\) would show the absence of the deduction proof term \(j_3\)
(introduced on line 5). This is a general phenomenon in CDSAT: first-order decisions play a role in finding models, not proofs; thus, deduction proof terms referring to first-order decisions cannot appear in the final proof term, and an easy optimization is to avoid constructing them.

5 From Proof Terms to Proofs

The CDSAT proof terms record enough information to reconstruct a proof of unsatisfiability from a CDSAT derivation concluding unsat. The proof system in Fig. 4 describes how proof terms specify the proof reconstruction steps to be performed. A deduction proof term \( j \) with \( H \vdash j : A \) indicates how to reconstruct a proof of \( H \models A \). A conflict proof term \( c \) with \( E \vdash c : \bot \) indicates how to reconstruct a proof of \( E \models \bot \). For instance, the rule for \( \text{res}(j, ^*A,c) \) can be read as: if \( j \) allows the reconstruction of a proof that \( A \) follows from \( H \), and if \( c \) allows the reconstruction of a proof that \( E \), \(^*A\) is unsatisfiable, then \( \text{res}(j, ^*A,c) \) allows the reconstruction of a proof that \( E \cup H \) is unsatisfiable. A solver implementing the proof-carrying CDSAT transition system can reconstruct the proof in any proof format that (1) features two kinds of proofs (deduction proofs and conflict proofs), and (2) is equipped with proof operations corresponding to the rules of Fig. 4. In this view, proof reconstruction interprets proof terms in the desired proof format.

An alternative option is that the CDSAT solver manipulates directly the proofs denoted by the expressions \( \text{in}(A), j_k, \text{lem}(H,c), \text{cfl}(j_k, a) \), and \( \text{res}(j, ^*A,c) \). The choice between the two approaches depends on whether the proofs generated by proof reconstruction are better (e.g., shorter, more readable) than the proof terms. Another factor is whether direct manipulation during the derivation causes such high runtime overhead that proof reconstruction is preferable. A third possibility is to consider proof terms themselves as the proof objects to be produced. Then a proof-checker would check directly the proof terms, checking that the rules of Fig. 4 are applied correctly. In the rest of this section, we see how CDSAT proof terms can be turned into resolution-style proofs (Subsection 5.1), and how the LCF approach to proofs can be applied to CDSAT (Subsection 5.2).

5.1 Proof Format Based on Resolution

Resolution proofs are based on clausal form. A few caveats are in order. First, CDSAT views logical connectives such as \( \land \) and \( \lor \) as interpreted symbols of the Boolean theory. Second, the existence of assignments such as \( (l_1 \lor l_2) \leftarrow \text{false} \) implies that clauses \( l_1 \lor \cdots \lor l_m \) may not be the only formulae in proofs. Third, CDSAT does not assume that its input is in Conjunctive Normal Form (CNF). Therefore, we distinguish between object-level clauses, that are the clauses \( l_1 \lor \cdots \lor l_m \), and CDSAT clauses, that are the clauses in resolution proofs reconstructed from CDSAT proof terms. A CDSAT clause, or clause for short when there is no ambiguity, is a set of singleton Boolean assignments, written \( L_1 \cdots L_m \), to avoid confusion with object-level clauses. A CDSAT clause is a clause in the standard propositional sense, with CDSAT formula (i.e., terms of sort \( \text{prop} \) in place of Boolean variables (aka propositional atoms). Since in CDSAT there are first-order assignments, the resolution proofs reconstructed from CDSAT proof terms use guarded clauses, written \( H \rightarrow C \), where \( H \) is a set of first-order assignments and \( C \) is a CDSAT clause. Guarded or hypothetical clauses are used extensively in the literature (e.g., [9, 15, 16]).

A resolution proof reconstructed from a CDSAT proof term is a binary tree such that:

- Every leaf is labeled with either a guarded clause, called in this case theory lemma, or an input singleton assignment; and
- Every internal node is labeled with a guarded clause, and, together with its two children, forms an instance of one of the following rules:

\[
\begin{align*}
H_1 \rightarrow C & \mid L_1 \rightarrow C_1 \mid L_2 \rightarrow C_2 \\
H_1 \cup H_2 \rightarrow C_1 \mid C_2 \\
L \rightarrow C & \mid \overline{L} \\
H \rightarrow C & \mid A \rightarrow H \rightarrow C
\end{align*}
\]

The top rule is binary resolution, generalized to the presence of first-order assignments which play no role in the rule. In the bottom two rules, the left premise is a leaf labeled by an input singleton assignment: the left-hand rule can be seen as unit resolution; the right-hand rule covers the analogous case for a first-order assignment \( A \).

In order to illustrate theory lemmas, consider a clausal form \( L_0 \) of \( \{L_1, \ldots, L_m\} \). The Boolean theory has the following lemmas, that can label the leaves of resolution trees:

\[
\begin{align*}
\emptyset & \rightarrow \overline{L_0} \mid \cdots \mid \overline{L_m} \\
\emptyset & \rightarrow L_0 \mid \cdots \mid L_1 \\
\ldots
\end{align*}
\]

Since \( L_0 \) is a clausal form of \( \{L_1, \ldots, L_m\} \), \( \overline{L_0} \mid \cdots \mid \overline{L_m} \) and \( L_0 \mid L_1 \), \( 1 \leq i \leq m \), are propositional tautologies. The first lemma allows the transformation of an object-level clause, the assignment \( L_0 \), into a CDSAT clause:

\[
\begin{align*}
H \rightarrow C & \mid \overline{L_0} \mid \cdots \mid \overline{L_m} \\
H \rightarrow C & \mid L_0 \mid \cdots \mid L_1 \\
\ldots
\end{align*}
\]

Conversely, the other lemmas allow the reification of a CDSAT clause \( L_1 \mid \cdots \mid L_m \) into an object-level clause:

\[
\begin{align*}
\emptyset & \rightarrow \overline{L_0} \mid \cdots \mid \overline{L_m} \\
\emptyset & \rightarrow L_0 \mid \cdots \mid L_1 \\
\ldots
\end{align*}
\]

H \rightarrow C \mid L_0

These transformations can be expressed compactly as the derivable inference rules
that involve the Boolean theory lemmas only if \( m \geq 2 \) (i.e., the clause is not a unit clause).

Given an assignment \( H \), we write \( H_{\text{FO}} \) for the greatest subset of \( H \) made of first-order assignments; and we write \( H_{\text{clause}} \) for the clause \( L_1|\cdots|L_m \) where \( L_1, \ldots, L_m \) are the singleton Boolean assignments in \( H \).

The resolution proof format for CDSAT interprets:
- A deduction proof term \( \emptyset \vdash \text{in}(A) : A \) as a proof concluding \( A \);
- A deduction proof term \( H \vdash j : L \) of another form as a proof concluding \( H_{\text{FO}} \rightarrow H_{\text{clause}} | L \);
- A conflict proof term \( H \vdash c : \bot \) as a proof concluding \( H_{\text{FO}} \rightarrow H_{\text{clause}} \).

The interpretation is defined by induction on proof terms as shown in Fig. 7. For instance, consider the interpretations of a deduction proof term and a conflict proof term that encapsulate a unit propagation, where \( K \) is a clausal form of \( H = H' \cup \{ L \} \) as in inference (1) from Section 3:

\[
\begin{align*}
\{ \{ \emptyset \} \cup H' \vdash \text{UP}(a, \{ a_1, \ldots, a_n \}) : \top \} & \\
\{ \{ \emptyset \} \cup H' \vdash \text{cfl}(\text{UP}(a, \{ a_1, \ldots, a_n \}), a_0) : \bot \}
\end{align*}
\]

where \( a_1, \ldots, a_n \) are the identifiers of the elements of \( H' \). Both become an instance of a Boolean theory lemma (2):

\[
\emptyset \vdash \neg K \quad H'_{\text{clause}} \quad \bot
\]

Since a CDSAT answer of the form unsat(c) means \( \emptyset \vdash c : \bot \), the resolution proof reconstructed from proof term \( c \) is a refutation, as its conclusion is \( \emptyset \rightarrow \emptyset \), that is, the empty clause. As already noted in Section 4, first-order decisions do not appear in proofs. It follows that first-order assignments may appear in proofs only if they are initial assignments. If the input problem contains no first-order assignments (i.e., it is an SMT problem), the reconstructed proof involves only singleton Boolean assignments labeling leaves and guarded clauses of the form \( \emptyset \rightarrow C \) labeling leaves or internal nodes. In other words, the reconstructed proof is a resolution refutation in the standard sense, deriving the empty clause, and with leaves labeled by input assignments or theory lemmas.

The use of theory lemmas for propositional reasoning is less usual and it descends from the choice of treating propositional logic as a theory. Advantages include the possibility of applying CDSAT to problems that are not in CNF, and the possibility of transforming object-level clauses into CDSAT clauses and vice versa, as shown above. These transformations also provide a native feature for sharing resolution proofs. For instance in the refutation of Section 4.4, the conflict proof term \( c_{10} = \text{res}(j_1, a_1(\neg l_2 \lor \neg l_4), c_{18}) \), with \( j_1 = \text{lem}(\{ a_1, a_2, a_4 \}) \), yields \( a_1 L_1, a_2 L_2, c_{19} : \bot \), where \( L_1 \) is \( \neg l_2 \lor \neg l_4 \) and \( L_2 \) is \( \neg l_2 \lor \neg l_4 \). The resolution proof reconstructed from \( c_{19} \) is:

\[
\begin{align*}
\neg c_{10} L_1, a_1 L_2, a_2 L_2, a_4 L_2, c_{19} : \bot & \\
\neg c_{10} L_1 | L_2 | (\neg l_2 \lor \neg l_4) & \\
\neg c_{10} L_1 | L_2 | (\neg l_2 \lor \neg l_4) & \\
\emptyset \rightarrow \neg c_{10} L_1
\end{align*}
\]

The fact that the learned clause \( \neg l_2 \lor \neg l_4 \) is reused in \( c_{18} \), as described in Section 4.4, means that the resolution proof \( \neg c_{10} L_1 | L_2 | (\neg l_2 \lor \neg l_4) \) has two leaves labeled by the same theory lemma:

\[
\emptyset \rightarrow \neg c_{10} L_1
\]

An alternative resolution proof with root \( \emptyset \rightarrow L_1 | L_2 \) can be obtained by replacing those two leaves with the subproof interpreting \( c_{2} \), and replacing \( \neg c_{10} L_1 | L_2 \) by \( L_1 | L_2 \) in all nodes underneath. This avoids the explicit conversions between the object-level clause \( \neg l_2 \lor \neg l_4 \) and the CDSAT clause \( L_2 | L_2 \). However, in this alternative proof, the subtree interpreting \( c_{2} \) is duplicated. Such duplications happen in the standard format of resolution trees where there is only one kind of clauses. By distinguishing between object-level clauses and CDSAT clauses, and using the former for input clauses and
the latter for generated clauses, CDSAT natively supports the explicit sharing of subproofs as in resolution DAG’s.

5.2 An LCF Architecture for CDSAT

Another example of proof format is the “dummy” one, where proofs do not contain any information other than what they are supposed to be the proofs of:

1. A deduction proof \( \Gamma \vdash \phi \) is the pair \( \langle \Gamma, \phi \rangle \), and
2. A conflict proof \( \phi \vdash \bot \) is \( H \).

Although this “proof format” does not allow any proof checking, the trustworthiness of a reasoner producing such proofs can still be established by the LCF programming abstraction [17, 30]. This approach uses a type theorem, whose constructed inhabitants are provable formulae. Actually, this type is defined as an alias for the type formula of formulae, but this is known only to a fixed and well-identified piece of code, called the LCF kernel. This kernel hides the definition of theorem to the outside world and exports a range of kernel primitives to manipulate inhabitants of type theorem in a safe and provably correct way. For instance, a primitive

modus_ponens : theorem -> theorem -> theorem

takes as arguments two inhabitants \( F \) and \( G \) of type theorem, checks that \( F \) is of the form \( G \Rightarrow R \), and returns \( R \) as an inhabitant of theorem. The kernel can export a primitive that reveals that any inhabitant of theorem is a formula, but obviously not one that casts any inhabitant of formula into type theorem. In this way, the existence of an inhabitant \( F \) of theorem witnesses the fact that \( F \) is provable, as an inhabitant only results from a series of correct manipulations by the kernel primitives: if the kernel code is trusted, then \( F \) can be trusted to be a theorem, while no proof has ever been constructed in memory.

CDSAT is well-suited for the LCF approach: given a type assign for assignments and single_assign for singleton assignments, a trusted kernel defines types

<table>
<thead>
<tr>
<th>type deduction</th>
<th>type conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td>in : single_assign -&gt; deduction</td>
<td>in : single_assign</td>
</tr>
<tr>
<td>coerc : 'k theory_handler -&gt; 'k theory_proof -&gt; deduction</td>
<td>coerc : 'k theory_handler</td>
</tr>
<tr>
<td>lem : conflict -&gt; assign -&gt; deduction</td>
<td>lem : conflict -&gt; assign</td>
</tr>
<tr>
<td>cfl : 'k theory_handler -&gt; conflict</td>
<td>cfl : 'k theory_handler</td>
</tr>
<tr>
<td>res : deduction -&gt; conflict -&gt; conflict</td>
<td>res : deduction</td>
</tr>
<tr>
<td>reveal : conflict -&gt; assign</td>
<td>reveal : conflict</td>
</tr>
</tbody>
</table>

Figure 8. API exported by a CDSAT kernel

a matching \( k \), as implemented for example by a Generalized Algebraic Data Type (GADT) [11, 37]. The handler allows the primitives to check that \( T_k \) is one of the combined theories before coercing the theory proof, trusted to be correct, into a deduction or a conflict. Proof-carrying CDSAT can be programmed on top of this kernel, so that, when it halts with answer unsat(c), the proof term \( c \) is an inhabitant of type conflict. The reveal primitive applied to \( c \) will return the empty assignment. Although no proof has been constructed in memory, the answer is correct by construction.

6 Discussion

Conflict-driven satisfiability procedures work by constructing partial assignments, detecting conflicts when the assignment falsifies the input formula, and performing conflict-driven inferences to learn new formulae and reorient the search. In prior work, we presented CDSAT as a combination procedure for conflict-driven theory satisfiability solvers for mutually disjoint theories, and proved its termination, soundness, and completeness under suitable assumptions. In the present work, we extend the CDSAT transition system with lemmatization so that new learned clauses can be added during backjumping from a conflict even in the context of a partial assignment. We also annotate the transition system in order to construct a proof object when the procedure discovers that the input problem is unsatisfiable. These proof objects can be rendered in a number of proof formats and the resulting proofs checked by a trusted checker or shown to be correct by construction.

In future work, we plan to implement a reasoner based on the CDSAT approach exploring different search plans and proof formats. Search plans play a key role in coordinating the work of theory modules, including prioritizing them with respect to both decisions and deductions. Topics for further investigations include extensions of proofs to account for preprocessing techniques widely used in SAT and SMT solving, evaluating the cost of proof generation and proof checking in CDSAT, and studying proof formats that reduce it as done for instance in SAT solving [13].
References


