ADCL: Acceleration Driven Clause Learning for Constrained Horn Clauses*

Florian Frohn(✉) and Jürgen Giesl(✉)

LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

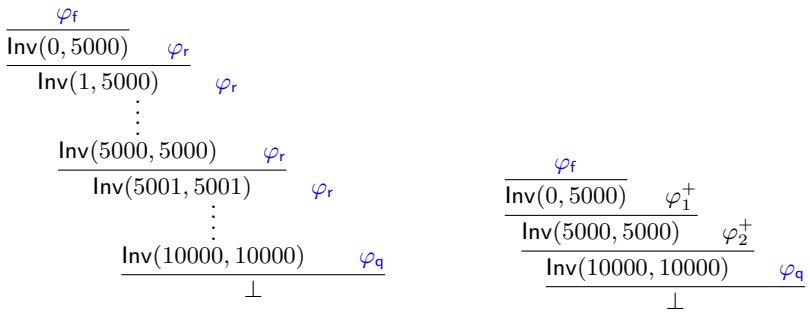
Abstract. *Constrained Horn Clauses* (CHCs) are often used in automated program verification. Thus, techniques for (dis-)proving satisfiability of CHCs are a very active field of research. On the other hand, *acceleration techniques* for computing formulas that characterize the N-fold closure of loops have successfully been used for static program analysis. We show how to use acceleration to avoid repeated derivations with recursive CHCs in resolution proofs, which reduces the length of the proofs drastically. This idea gives rise to a novel calculus for (dis)proving satisfiability of CHCs, called *Acceleration Driven Clause Learning (ADCL)*. We implemented this new calculus in our tool LoAT and evaluate it empirically in comparison to other state-of-the-art tools.

1 Introduction

Constrained Horn Clauses (CHCs) are often used for expressing verification conditions in automated program verification. Examples for tools based on CHCs include Korn [19] and SeaHorn [30] for verifying C and C++ programs, JayHorn for Java programs [36], HornDroid for Android apps [12], RustHorn for Rust programs [43], and SmartACE [50] and SolCMC [3] for Solidity. Consequently, techniques for (dis-)proving satisfiability of CHCs (CHC-SAT) are a very active field of research, resulting in powerful tools like Spacer [37], Eldarica [35], FreqHorn [20], Golem [7], Ultimate [17], and RInGEN [38].

On the other hand, *loop acceleration techniques* have been used successfully for static program analyses during the last years, resulting in tools like Flata [9,27] and LoAT [24]. Essentially, such techniques compute quantifier-free first-order formulas that characterize the N-fold closure of the transition relation of loops without branching in their body. Thus, acceleration techniques can be used when generating verification conditions in order to replace such loops with the closure of their transition relation.

* funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2)



(a) Original Resolution Proof (b) Accelerated Resolution Proof

Fig. 1: Original and Accelerated Resolution Proof

In this paper, we apply acceleration techniques to CHC-SAT, where we restrict ourselves to linear CHCs, i.e., clauses that contain at most one positive and one negative literal with uninterpreted predicates. As our main interest lies in proving *un*satisfiability of CHCs, our approach does not rely on abstractions, in contrast to most other techniques. Instead, we use acceleration to cut off repeated derivations with recursive CHCs while exploring the state space via resolution. In this way, the number of resolution steps that are required to reach a counterexample can be reduced drastically, as new CHCs that are "learned" via acceleration can simulate arbitrarily many "ordinary" resolution steps at once.

Example 1. Consider the following set of CHCs Φ over the theory of linear integer arithmetic (LIA) with a *fact* φ_f, a *rule* φ_r, and a *query* φ_q, where $\top$ and $\bot$ stand for *true* and *false*:[1]

$$\begin{aligned} \top &\implies \mathsf{Inv}(0, 5000) && (\varphi_f)\\ \mathsf{Inv}(X_1, X_2) \wedge & \\ ((X_1 < 5000 \wedge Y_2 = X_2) \vee (X_1 \geq 5000 \wedge Y_2 = X_2 + 1)) &\implies \mathsf{Inv}(X_1 + 1, Y_2) && (\varphi_r)\\ \mathsf{Inv}(10000, 10000) &\implies \bot && (\varphi_q) \end{aligned}$$

Its unsatisfiability can be proven via resolution and arithmetic simplification as shown in Fig. 1a. The proof requires 10001 resolution steps. Using acceleration techniques, we can derive the following two new CHCs from Φ:

$$\begin{aligned} \mathsf{Inv}(X_1, X_2) \wedge N > 0 \wedge X_1 + N < 5001 &\implies \mathsf{Inv}(X_1 + N, X_2) && (\varphi_1^+)\\ \mathsf{Inv}(X_1, X_2) \wedge N > 0 \wedge X_1 \geq 5000 &\implies \mathsf{Inv}(X_1 + N, X_2 + N) && (\varphi_2^+) \end{aligned}$$

The first CHC φ_1^+ covers arbitrarily many subsequent resolution steps with φ_r where $X_1 < 5000$. Similarly, the second CHC φ_2^+ covers arbitrarily many steps where $X_1 \geq 5000$. Now we can prove unsatisfiability of Φ with just 3 resolution steps, as shown in Fig. 1b.

[1] `chc-LIA-Lin_052.smt2` from the benchmarks of the CHC Competition '22 [14]

This idea gives rise to a novel calculus for CHC-SAT, called *Acceleration Driven Clause Learning (ADCL)*. ADCL is refutationally complete and can also prove satisfiability, but it does not necessarily terminate.

So far, our implementation in our tool LoAT is restricted to proving unsatisfiability. In program verification (which is one of the most important applications of CHC-SAT), satisfiability usually corresponds to safety, i.e., if an error state is reachable in the original program, then the CHCs derived from the program are unsatisfiable. Hence, LoAT can be used to show reachability of error states in program verification. The "witness" of reachability is a resolution proof ending in a conditional empty clause $\psi \implies \bot$ (where the condition ψ is a formula over the signature of some background theory like LIA), together with a model for ψ. Instantiating the variables in the proof according to the model yields a proof on ground instances. Then this instantiated proof corresponds to a program run that ends in an error state, i.e., a counterexample.

After introducing preliminaries in Sect. 2, we formalize ADCL in Sect. 3. Next, we discuss how to implement ADCL efficiently in Sect. 4. In Sect. 5, we discuss related work, and we show that our approach is highly competitive with state-of-the-art CHC-SAT solvers by means of an empirical evaluation. All proofs can be found in the extended version [26].

2 Preliminaries

We assume that the reader is familiar with basic concepts from many-sorted first-order logic. Throughout this paper, Σ denotes a many-sorted first-order signature that just contains predicates, i.e., we do not consider uninterpreted functions. Moreover, $\mathcal{V}$ denotes a countably infinite set of variables and for each entity e, $\mathcal{V}(e)$ denotes the variables occurring in e. We write $\vec{x}$ for sequences and x_i is the i^{th} element of $\vec{x}$. In the following, we introduce preliminaries regarding *Constrained Horn Clauses* and *acceleration techniques*.

Constrained Horn Clauses are first-order formulas of the form

$$\forall \vec{X}_1, \ldots, \vec{X}_d, \vec{Y}, \vec{Z}.\ \mathsf{F}_1(\vec{X}_1) \wedge \ldots \wedge \mathsf{F}_d(\vec{X}_d) \wedge \psi \implies \mathsf{G}(\vec{Y}) \qquad \text{or}$$
$$\forall \vec{X}_1, \ldots, \vec{X}_d, \vec{Z}.\ \mathsf{F}_1(\vec{X}_1) \wedge \ldots \wedge \mathsf{F}_d(\vec{X}_d) \wedge \psi \implies \bot$$

where $\vec{X}_1, \ldots, \vec{X}_d, \vec{Y}, \vec{Z}$ are pairwise disjoint vectors of pairwise different variables, $\mathsf{F}_1, \ldots, \mathsf{F}_d, \mathsf{G} \in \Sigma$, $\psi \in \mathsf{QF}(\mathcal{A})$ is a quantifier-free first-order formula over the many-sorted signature $\Sigma_{\mathcal{A}}$ of some theory $\mathcal{A}$, and $\mathcal{V}(\psi) \subseteq \vec{X}_1 \cup \ldots \cup \vec{X}_d \cup \vec{Y} \cup \vec{Z}$. We assume that $\mathcal{A}$ is a complete theory with equality and that Σ and $\Sigma_{\mathcal{A}}$ are disjoint, and we usually omit the leading universal quantifier, i.e., all variables in CHCs are implicitly universally quantified.[2] Moreover, w.l.o.g., we assume that

[2] We assume that all arguments of predicates are variables. This is not a restriction, as one can add equations to ψ to identify $\Sigma_{\mathcal{A}}$-terms with fresh variables. To ease the presentation, we also use $\Sigma_{\mathcal{A}}$-terms as arguments of predicates in examples (e.g., in Ex. 1 we wrote $\top \implies \mathsf{Inv}(0, 5000)$ instead of $Y_1 = 0 \wedge Y_2 = 5000 \implies \mathsf{Inv}(Y_1, Y_2)$).

ψ is in negation normal form. In this paper, we restrict ourselves to *linear* CHCs. Thus, we consider CHCs of the following form:

$$\psi \implies \mathsf{G}(\vec{Y}) \quad \text{(fact)} \qquad \mathsf{F}(\vec{X}) \wedge \psi \implies \bot \quad \text{(query)}$$
$$\mathsf{F}(\vec{X}) \wedge \psi \implies \mathsf{G}(\vec{Y}) \quad \text{(rule)} \qquad \psi \implies \bot \quad \text{(conditional empty clause)}$$

The premise and conclusion of a CHC is also called *body* and *head*, a CHC is *recursive* if it is a rule where $\mathsf{F} = \mathsf{G}$, and it is *conjunctive* if ψ is a conjunction of literals. Throughout this paper, φ and π always denote CHCs. The *condition* of a CHC φ is $\mathsf{cond}(\varphi) := \psi$. We write $\varphi|_{\psi'}$ for the CHC that results from φ by replacing $\mathsf{cond}(\varphi)$ with ψ'. Typically, the original set of CHCs does not contain conditional empty clauses, but in our setting, such clauses result from resolution proofs that start with a fact and end with a query. A conditional empty clause is called a *refutation* if its condition is satisfiable. We also refer to sets of CHCs as *CHC problems*, denoted Φ or Π.

We call σ an $\mathcal{A}$-*interpretation* if it is a model of $\mathcal{A}$ whose carrier only contains ground terms over $\Sigma_{\mathcal{A}}$, extended with interpretations for Σ and $\mathcal{V}$. Given a first-order formula η over $\Sigma \cup \Sigma_{\mathcal{A}}$, an $\mathcal{A}$-interpretation σ is a *model* of η (written $\sigma \models_{\mathcal{A}} \eta$) if it satisfies η. If such a model exists, then η is *satisfiable*. As usual, $\models_{\mathcal{A}} \eta$ means that η is valid (i.e., we have $\sigma \models_{\mathcal{A}} \eta$ for all $\mathcal{A}$-interpretations σ) and $\eta \equiv_{\mathcal{A}} \eta'$ means $\models_{\mathcal{A}} \eta \iff \eta'$. For sets of formulas H, we define $\sigma \models_{\mathcal{A}} H$ if $\sigma \models_{\mathcal{A}} \bigwedge_{\eta \in H} \eta$. The *ground instances* of a CHC $\eta \wedge \psi \implies \eta'$ are:

$$\mathsf{grnd}(\eta \wedge \psi \implies \eta') := \{\eta\sigma \implies \eta'\sigma \mid \sigma \models_{\mathcal{A}} \psi\},$$

where $\eta\sigma$ abbreviates $\sigma(\eta)$, i.e., it results from η by instantiating all variables according to σ. Since $\mathcal{A}$ is complete (i.e., either $\models_{\mathcal{A}} \psi$ or $\models_{\mathcal{A}} \neg\psi$ holds for every closed formula ψ over $\Sigma_{\mathcal{A}}$), $\mathcal{A}$-interpretations σ only differ on Σ and $\mathcal{V}$, and thus we have $\sigma \models_{\mathcal{A}} \varphi$ iff $\sigma \models_{\mathcal{A}} \mathsf{grnd}(\varphi)$.

In the following, we use "::" for the concatenation of sequences, where we identify sequences of length 1 with their elements, i.e., we sometimes write $x :: xs$ instead of $[x] :: xs$ or $x :: y$ instead of $[x, y]$. As usual, $\mathsf{mgu}(s, t)$ is the most general unifier of s and t. The following definition formalizes resolution (where we disregard the underlying theory and just use ordinary syntactic unification). If the corresponding literals of two clauses φ, φ' do not unify, then we define their resolvent to be $\bot \implies \bot$, so that resolution is defined for pairs of arbitrary CHCs. Note that in our setting, the mgu θ is always a variable renaming.

Definition 2 (Resolution). *Let φ and φ' be CHCs, where we rename the variables in φ and φ' such that they are disjoint. If*

$$\varphi = (\eta \wedge \psi \implies \mathsf{F}(\vec{x})), \quad \varphi' = (\mathsf{F}(\vec{y}) \wedge \psi' \implies \eta'), \quad \textit{and} \quad \theta = \mathsf{mgu}(\mathsf{F}(\vec{x}), \mathsf{F}(\vec{y})),$$
$$\textit{then} \qquad \mathsf{res}(\varphi, \varphi') := (\eta \wedge \psi \wedge \psi' \implies \eta')\theta.$$
$$\textit{Otherwise,} \qquad \mathsf{res}(\varphi, \varphi') := (\bot \implies \bot).$$

Here, η can also be $\top$ and η' can also be $\bot$. We lift res *to non-empty sequences of CHCs by defining*

$$\mathsf{res}([\varphi_1, \varphi_2] :: \vec{\varphi}) := \mathsf{res}(\mathsf{res}(\varphi_1, \varphi_2) :: \vec{\varphi}) \qquad \textit{and} \qquad \mathsf{res}(\varphi_1) := \varphi_1.$$

We implicitly lift notations and terminology for CHCs to sequences of CHCs via resolution. So for example, we have $\mathsf{cond}(\vec{\varphi}) := \mathsf{cond}(\mathsf{res}(\vec{\varphi}))$ and $\mathsf{grnd}(\vec{\varphi}) := \mathsf{grnd}(\mathsf{res}(\vec{\varphi}))$.

Example 3. Consider a variation Φ' of the CHC problem Φ from Ex. 1 where φ_f is replaced by

$$X_1 \leq 0 \wedge X_2 \geq 5000 \implies \mathsf{Inv}(X_1, X_2) \qquad (\varphi'_\mathsf{f})$$

To prove its unsatisfiability, one can consider the resolvent of the sequence $\vec{\varphi} := [\varphi'_\mathsf{f}, \varphi_1^+, \varphi_2^+, \varphi_\mathsf{q}]$:

$$\begin{aligned}
& \mathsf{res}([\varphi'_\mathsf{f}, \varphi_1^+, \varphi_2^+, \varphi_\mathsf{q}]) \\
= \;& \mathsf{res}([\psi \implies \mathsf{Inv}(X_1 + N, X_2), \varphi_2^+, \varphi_\mathsf{q}]) \\
= \;& \mathsf{res}([\psi \wedge N' > 0 \wedge X_1{+}N \geq 5000 \implies \mathsf{Inv}(X_1{+}N{+}N', X_2{+}N'), \varphi_\mathsf{q}]) \quad (\dagger) \\
\equiv_\mathcal{A} \;& \mathsf{res}([X_1 \leq 0 \wedge X_2 \geq 5000 \wedge N' > 0 \implies \mathsf{Inv}(5000 + N', X_2 + N'), \varphi_\mathsf{q}]) \quad (\ddagger) \\
= \;& (X_1 \leq 0 \wedge X_2 \geq 5000 \wedge N' > 0 \wedge 5000 + N' = X_2 + N' = 10000 \implies \bot)
\end{aligned}$$

Here, we have

$$\psi := \mathsf{cond}([\varphi'_\mathsf{f}, \varphi_1^+]) = X_1 \leq 0 \wedge X_2 \geq 5000 \wedge N > 0 \wedge X_1 + N < 5001.$$

In the step marked with (†), the variable N' results from renaming N in φ_2^+. In the step marked with (‡), we simplified $X_1 + N$ to 5000 for readability, as $\psi \wedge X_1 + N \geq 5000$ implies $X_1 + N = 5000$. If $\sigma(X_1) = 0$ and $\sigma(X_2) = \sigma(N') = 5000$, then $\sigma \models_\mathcal{A} \mathsf{res}(\vec{\varphi})$ and thus $\mathsf{res}(\vec{\varphi})$ is a refutation, so Φ is unsatisfiable. By instantiating the variables in the proof according to σ (and setting N to 5000, as we had $X_1 + N = 5000$), we obtain an accelerated resolution proof on ground instances that is analogous to Fig. 1b and serves as a "witness" of unsatisfiability, i.e., a "counterexample" to Φ'.

Acceleration Techniques are used to compute the N-fold closure of the transition relation of a loop in program analysis. In the context of CHCs, applying an acceleration technique to a recursive CHC φ yields another CHC φ' which, for any instantiation of a dedicated fresh variable $N \in \mathcal{V}(\varphi')$ with a positive integer, has the same ground instances as $\mathsf{res}(\varphi^N)$. Here, φ^N denotes the sequence consisting of N repetitions of φ. In the following definition, we restrict ourselves to conjunctive CHCs, since many existing acceleration techniques do not support disjunctions [8], or have to resort to approximations in the presence of disjunctions [23].

Definition 4 (Acceleration). *An* acceleration technique *is a function* accel *that maps a recursive conjunctive CHC* φ *to a recursive conjunctive CHC such that* $\mathsf{grnd}(\mathsf{accel}(\varphi)) = \bigcup_{n \in \mathbb{N}_{\geq 1}} \mathsf{grnd}(\varphi^n)$.

Example 5. In the CHC problem from Ex. 1, φ_r entails

$$\mathsf{Inv}(X_1, X_2) \wedge X_1 < 5000 \implies \mathsf{Inv}(X_1 + 1, X_2).$$

From this CHC, an acceleration technique would compute φ_1^+.

Note that most theories are not "closed under acceleration". For example, consider the left clause below, which only uses linear arithmetic.

$$\mathsf{F}(X,Y) \implies \mathsf{F}(X+Y,Y) \qquad\qquad \mathsf{F}(X,Y) \wedge N > 0 \implies \mathsf{F}(X+N\cdot Y,Y)$$

Accelerating it yields the clause on the right, which is not expressible with linear arithmetic due to the sub-expression $N \cdot Y$. Moreover, if there is no sort for integers in the background theory $\mathcal{A}$, then an additional sort for the range of N is required in the formula that results from applying accel. For that reason, we consider *many-sorted* first-order logic and theories.

3 Acceleration Driven Clause Learning

In this section, we introduce our novel calculus ADCL for (dis)proving satisfiability of CHC problems. In Sect. 3.1, we start with important concepts that ADCL is based on. Then the ADCL calculus itself is presented in Sect. 3.2. Finally, in Sect. 3.3 we investigate the main properties of ADCL.

3.1 Syntactic Implicants and Redundancy

Since ADCL relies on acceleration techniques, an important property of ADCL is that it only applies resolution to conjunctive CHCs, even if the analyzed CHC problem is not conjunctive. To obtain conjunctive CHCs from non-conjunctive CHCs, we use *syntactic implicants*.

Definition 6 (Syntactic Implicant Projection). *Let $\psi \in \mathsf{QF}(\mathcal{A})$ be in negation normal form. We define:*

$$\begin{aligned}
\mathsf{sip}(\psi,\sigma) &:= \bigwedge\{\ell \text{ is a literal of } \psi \mid \sigma \models_{\mathcal{A}} \ell\} && \text{if } \sigma \models_{\mathcal{A}} \psi\\
\mathsf{sip}(\psi) &:= \{\mathsf{sip}(\psi,\sigma) \mid \sigma \models_{\mathcal{A}} \psi\}\\
\mathsf{sip}(\varphi) &:= \{\varphi|_\psi \mid \psi \in \mathsf{sip}(\mathsf{cond}(\varphi))\} && \text{for CHCs } \varphi\\
\mathsf{sip}(\Phi) &:= \bigcup_{\varphi\in\Phi} \mathsf{sip}(\varphi) && \text{for sets of CHCs } \Phi
\end{aligned}$$

Here, sip *abbreviates* syntactic implicant projection.

In contrast to the usual notion of implicants (which just requires that the implicants entail ψ), syntactic implicants are restricted to literals from ψ to ensure that $\mathsf{sip}(\psi)$ is finite. We call such implicants *syntactic* since Def. 6 does not take the semantics of literals into account. For example, the formula $\psi := (X > 0 \wedge X > 1)$ contains the literals $X > 0$ and $X > 1$, and $\models_{\mathcal{A}} X > 1 \implies \psi$, but $(X > 1) \notin \mathsf{sip}(\psi) = \{\psi\}$, because every model of $X > 1$ also satisfies $X > 0$. It is easy to show that $\psi \equiv_{\mathcal{A}} \bigvee \mathsf{sip}(\psi)$, and thus we also have $\Phi \equiv_{\mathcal{A}} \mathsf{sip}(\Phi)$.

Example 7. In the CHC problem of Ex. 1, we have

$$\mathsf{sip}(\mathsf{cond}(\varphi_{\mathsf{r}})) = \{(X_1 < 5000 \wedge Y_2 = X_2), (X_1 \geq 5000 \wedge Y_2 = X_2 + 1)\}.$$

Since $\mathsf{sip}(\varphi)$ is worst-case exponential in the size of $\mathsf{cond}(\varphi)$, we do not compute it explicitly: When resolving with φ, we conjoin $\mathsf{cond}(\varphi)$ to the condition of the

resulting resolvent and search for a model σ. This ensures that we do not continue with resolvents that have unsatisfiable conditions. Then we replace $\mathsf{cond}(\varphi)$ by $\mathsf{sip}(\mathsf{cond}(\varphi), \sigma)$ in the resolvent. This corresponds to a resolution step with a conjunctive variant of φ whose condition is satisfied by σ. In other words, our calculus constructs $\mathsf{sip}(\mathsf{cond}(\varphi), \sigma)$ "on the fly" when resolving $\vec{\varphi}$ with φ, where $\sigma \models_{\mathcal{A}} \mathsf{cond}(\vec{\varphi} :: \varphi)$, see Sect. 4 for details. In this way, the exponential blowup that results from constructing $\mathsf{sip}(\varphi)$ explicitly can often be avoided.

As ADCL learns new clauses via acceleration, it is important to prefer more general (learned) clauses over more specific clauses in resolution proofs. To this end, we use the following *redundancy relation* for CHCs.

Definition 8 (Redundancy Relation). *For two CHCs φ and π, we say that φ is* (strictly) redundant *w.r.t. π, denoted $\varphi \sqsubseteq \pi$ ($\varphi \sqsubset \pi$), if $\mathsf{grnd}(\varphi) \subseteq \mathsf{grnd}(\pi)$ ($\mathsf{grnd}(\varphi) \subset \mathsf{grnd}(\pi)$). For a set of CHCs Π, we define $\varphi \sqsubseteq \Pi$ ($\varphi \sqsubset \Pi$) if $\varphi \sqsubseteq \pi$ ($\varphi \sqsubset \pi$) for some $\pi \in \Pi$.*

In the following, we assume that we have oracles for checking redundancy, for satisfiability of $\mathsf{QF}(\mathcal{A})$-formulas, and for acceleration. In practice, we have to resort to incomplete techniques instead. In Sect. 4, we will explain how our implementation takes that into account.

3.2 The ADCL Calculus

A *state* of ADCL consists of a CHC problem Π, containing the original CHCs and all *learned* clauses that were constructed by acceleration, the *trace* $[\varphi_i]_{i=1}^k$, representing a resolution proof, and a sequence $[B_i]_{i=0}^k$ of sets of *blocking clauses*. Clauses $\varphi \sqsubseteq B_i$ must not be used for the $(i+1)^{th}$ resolution step. In this way, blocking clauses prevent ADCL from visiting the same part of the search space more than once. ADCL blocks a clause φ after proving that $\bot$ (and thus unsat) cannot be derived after adding φ to the current trace, or if the current trace $\vec{\varphi} :: \vec{\varphi}'$ ends with φ and there is another "more general" trace $\vec{\varphi} :: \vec{\pi}$ such that $\vec{\varphi}' \sqsubseteq \vec{\pi}$ and $|\vec{\varphi}'| \geq |\vec{\pi}|$, where one of the two relations is strict. In the following, Φ denotes the original CHC problem whose satisfiability is analyzed with ADCL.

Definition 9 (State). *A* state *is a triple*

$$(\Pi, [\varphi_i]_{i=1}^k, [B_i]_{i=0}^k)$$

where $\Pi \supseteq \Phi$ is a CHC problem, $B_i \subseteq \mathsf{sip}(\Pi)$ for each $0 \leq i \leq k$, and $[\varphi_i]_{i=1}^k \in \mathsf{sip}(\Pi)^*$. The clauses in $\mathsf{sip}(\Phi)$ are called original clauses *and all clauses in $\mathsf{sip}(\Pi) \setminus \mathsf{sip}(\Phi)$ are called* learned clauses*. A clause $\varphi \sqsubseteq B_k$ is* blocked*, and φ is* active *if it is not blocked and $\mathsf{cond}([\varphi_i]_{i=1}^k :: \varphi)$ is satisfiable.*

Now we are ready to introduce our novel calculus.

Definition 10 (ADCL). *Let the "backtrack function" bt be defined as*

$$\mathsf{bt}(\Pi, [\varphi_i]_{i=1}^k, [B_0, \ldots, B_k]) := (\Pi, [\varphi_i]_{i=1}^{k-1}, [B_0, \ldots, B_{k-1} \cup \{\varphi_k\}]).$$

Our calculus is defined by the following rules.

$$\frac{}{\Phi \rightsquigarrow (\Phi, [], [\varnothing])} \quad (\textsc{Init})$$

$$\frac{\varphi \in \mathsf{sip}(\Pi) \text{ is active}}{(\Pi, \vec{\varphi}, \vec{B}) \rightsquigarrow (\Pi, \vec{\varphi} :: \varphi, \vec{B} :: \varnothing)} \quad (\textsc{Step})$$

$$\frac{\vec{\varphi}^{\circlearrowleft} \text{ is recursive} \quad |\vec{\varphi}^{\circlearrowleft}| = |\vec{B}^{\circlearrowleft}| \quad \mathsf{accel}(\vec{\varphi}^{\circlearrowleft}) = \varphi}{(\Pi, \vec{\varphi} :: \vec{\varphi}^{\circlearrowleft}, \vec{B} :: \vec{B}^{\circlearrowleft}) \rightsquigarrow (\Pi \cup \{\varphi\}, \vec{\varphi} :: \varphi, \vec{B} :: \{\varphi\})} \quad (\textsc{Accelerate})$$

$$\frac{\vec{\varphi}' \sqsubset \mathsf{sip}(\Pi) \quad \text{or} \quad \vec{\varphi}' \sqsubseteq \mathsf{sip}(\Pi) \wedge |\vec{\varphi}'| > 1}{s = (\Pi, \vec{\varphi} :: \vec{\varphi}', \vec{B}) \rightsquigarrow \mathsf{bt}(s)} \quad (\textsc{Covered})$$

$$\frac{\text{all rules and queries from } \mathsf{sip}(\Pi) \text{ are inactive} \quad \varphi \text{ is not a query}}{s = (\Pi, \vec{\varphi} :: \varphi, \vec{B}) \rightsquigarrow \mathsf{bt}(s)} \quad (\textsc{Backtrack})$$

$$\frac{\vec{\varphi} \text{ is a refutation}}{(\Pi, \vec{\varphi}, \vec{B}) \rightsquigarrow \mathsf{unsat}} \quad (\textsc{Refute})$$

$$\frac{\text{all facts and conditional empty clauses from } \mathsf{sip}(\Pi) \text{ are inactive}}{(\Pi, [], [B]) \rightsquigarrow \mathsf{sat}} \quad (\textsc{Prove})$$

We write $\overset{\mathrm{I}}{\rightsquigarrow}, \overset{\mathrm{S}}{\rightsquigarrow}, \ldots$ to indicate that Init, Step, ... was used for a $\rightsquigarrow$-step. All derivations start with Init. Step adds an active CHC φ to the trace. Due to the linearity of CHCs, we can restrict ourselves to proofs that start with a fact or a conditional empty clause, but such a restriction is not needed for the correctness of our calculus and thus not enforced.

As soon as $\vec{\varphi}$ has a recursive suffix $\vec{\varphi}^{\circlearrowleft}$ (i.e., a suffix $\vec{\varphi}^{\circlearrowleft}$ such that $\mathsf{res}(\vec{\varphi}^{\circlearrowleft})$ is recursive), Accelerate can be used. Then the suffix $\vec{\varphi}^{\circlearrowleft}$ is replaced by the accelerated clause φ and the suffix $\vec{B}^{\circlearrowleft}$ of sets of blocked clauses that corresponds to $\vec{\varphi}^{\circlearrowleft}$ is replaced by $\{\varphi\}$. The reason is that for learned clauses, we always have $\mathsf{res}(\varphi, \varphi) \sqsubseteq \varphi$, and thus applying φ twice in a row is superfluous. So in this way, clauses that were just learned are not used for resolution several times in a row. As mentioned in Sect. 2, the condition of the learned clause may not be expressible in the theory $\mathcal{A}$. Thus, when Accelerate is applied, we may implicitly switch to a richer theory $\mathcal{A}'$ (e.g., from linear to non-linear arithmetic).

If a suffix $\vec{\varphi}'$ of the trace is redundant w.r.t. $\mathsf{sip}(\Pi)$, we can backtrack via Covered, which removes the last element from $\vec{\varphi}'$ (but not the rest of $\vec{\varphi}'$, since this sequence could now be continued in a different way) and blocks it, such that we do not revisit the corresponding part of the search space. So here the redundancy check allows us to use more general (learned) clauses, if available. Here, it is important that we do not backtrack if $\vec{\varphi}'$ is a single, *weakly* redundant clause. Otherwise, Covered could always be applied after Step or Accelerate and block the last clause from the trace. Thus, we might falsely "prove" satisfiability.

If no further Step is possible since all CHCs are inactive, then we Backtrack as well and block the last clause from $\vec{\varphi}$ to avoid performing the same Step again.

If we started with a fact and the last CHC in $\vec{\varphi}$ is a query, then $\mathsf{res}(\vec{\varphi})$ is a refutation and Refute can be used to prove unsat.

Finally, if we arrive in a state where $\vec{\varphi}$ is empty and all facts and conditional empty clauses are inactive, then Prove is applicable as we have exhausted the entire search space without proving unsatisfiability, i.e., Φ is satisfiable. Note that we always have $|\vec{B}| = |\vec{\varphi}| + 1$, since we need one additional set of blocking clauses to block facts. While B_0 is initially empty (see Init), it can be populated via Backtrack or Covered. So eventually, all facts may become blocked, such that sat can be proven via Prove.

Example 11. Using our calculus, unsatisfiability of the CHC problem Φ in Ex. 1 can be proven as follows:

$$
\begin{aligned}
&\Phi \overset{\mathrm{I}}{\rightsquigarrow} (\Phi, [], [\varnothing]) \\
&\quad \overset{\mathrm{S}}{\rightsquigarrow} (\Phi, [\varphi_{\mathsf{f}}], [\varnothing, \varnothing]) && \top \implies \mathsf{Inv}(0, 5000) \\
&\quad \overset{\mathrm{S}}{\rightsquigarrow} (\Phi, [\varphi_{\mathsf{f}}, \varphi_{\mathsf{r}}|_{\psi_1}], [\varnothing, \varnothing, \varnothing]) && \mathsf{Inv}(0, 5000) \implies \mathsf{Inv}(1, 5000) \\
&\quad \overset{\mathrm{A}}{\rightsquigarrow} (\Pi_1, [\varphi_{\mathsf{f}}, \varphi_1^+], [\varnothing, \varnothing, \{\varphi_1^+\}]) && \mathsf{Inv}(0, 5000) \implies \mathsf{Inv}(1, 5000) \\
&\quad \overset{\mathrm{S}}{\rightsquigarrow} (\Pi_1, [\varphi_{\mathsf{f}}, \varphi_1^+, \varphi_{\mathsf{r}}|_{\psi_2}], [\varnothing, \varnothing, \{\varphi_1^+, \}, \varnothing]) && \mathsf{Inv}(5000, 5000) \implies \mathsf{Inv}(5001, 5001) \\
&\quad \overset{\mathrm{A}}{\rightsquigarrow} (\Pi_2, [\varphi_{\mathsf{f}}, \varphi_1^+, \varphi_2^+], [\varnothing, \varnothing, \{\varphi_1^+\}, \{\varphi_2^+\}]) && \mathsf{Inv}(5000, 5000) \implies \mathsf{Inv}(5001, 5001) \\
&\quad \overset{\mathrm{S}}{\rightsquigarrow} (\Pi_2, [\varphi_{\mathsf{f}}, \varphi_1^+, \varphi_2^+, \varphi_{\mathsf{q}}], [\varnothing, \varnothing, \{\varphi_1^+\}, \{\varphi_2^+\}, \varnothing]) && \mathsf{Inv}(10000, 10000) \implies \bot \\
&\quad \overset{\mathrm{R}}{\rightsquigarrow} \mathsf{unsat}
\end{aligned}
$$

Here, we have:

$$
\begin{aligned}
\Pi_1 &:= \Phi \cup \{\varphi_1^+\} & \psi_1 &:= X_1 < 5000 \wedge Y_2 = X_2 \\
\Pi_2 &:= \Pi_1 \cup \{\varphi_2^+\} & \psi_2 &:= X_1 \geq 5000 \wedge Y_2 = X_2 + 1
\end{aligned}
$$

Beside the state of our calculus, we show a ground instance of the last element φ of $\vec{\varphi}$ which results from applying a model for $\mathsf{cond}(\vec{\varphi})$ to φ. In our implementation, we always maintain such a model. In general, these models are not unique: For example, after the first acceleration step, we might use $\mathsf{Inv}(0, 5000) \implies \mathsf{Inv}(X_1, 5000)$ for any $X_1 \in [1, 5000]$. The reason is that φ_1^+ can simulate arbitrarily many resolution steps with $\varphi_{\mathsf{r}}|_{\psi_1}$, depending on the choice of N.

After starting the derivation with Init, we apply the only fact φ_{f} via Step. Next, we apply φ_{r}, projected to the case $X_1 < 5000$. Since φ_{r} is recursive, we may apply Accelerate afterwards, resulting in the new clause φ_1^+.

Then we apply φ_{r}, projected to the case $X_1 \geq 5000$. Note that the current model (resulting in the ground head-literal $\mathsf{Inv}(1, 5000)$) cannot be extended to a model for $\varphi_{\mathsf{r}}|_{\psi_2}$ (which requires $X_1 \geq 5000$). However, as the model is not part of the state, we may choose a different one at any point, which is important for implementing ADCL via *incremental* SMT, see Sect. 4. Hence, we can apply

$\varphi_{\mathsf{r}}|_{\psi_2}$ nevertheless.

Now we apply Accelerate again, resulting in the new clause φ_2^+. Finally, we apply the only query φ_{q} via Step, resulting in a conditional empty clause with a satisfiable condition, such that we can finish the proof via Refute.

Later (in Def. 14), we will define *reasonable strategies* for applying the rules of our calculus, which ensure that we use Accelerate instead of applying Step 10001 times in our example.

To see how our calculus proves satisfiability, assume that we replace φ_{q} with

$$\mathsf{Inv}(10000, X_2) \wedge X_2 \neq 10000 \implies \bot.$$

Then resolution with φ_{q} via the rule Step is no longer applicable and our derivation continues as follows after the second application of Accelerate:

$$\begin{aligned}
&(\Pi_2, [\varphi_{\mathsf{f}}, \varphi_1^+, \varphi_2^+], [\varnothing, \varnothing, \{\varphi_1^+\}, \{\varphi_2^+\}]) \\
\overset{\mathrm{B}}{\rightsquigarrow}\ &(\Pi_2, [\varphi_{\mathsf{f}}, \varphi_1^+], [\varnothing, \varnothing, \{\varphi_1^+, \varphi_2^+\}]) \\
\overset{\mathrm{B}}{\rightsquigarrow}\ &(\Pi_2, [\varphi_{\mathsf{f}}], [\varnothing, \{\varphi_1^+\}]) \\
\overset{\mathrm{B}}{\rightsquigarrow}\ &(\Pi_2, [], [\{\varphi_{\mathsf{f}}\}]) \\
\overset{\mathrm{P}}{\rightsquigarrow}\ &\mathsf{sat}
\end{aligned}$$

For all three Backtrack-steps, φ_{q} is clearly inactive, as adding it to $\vec{\varphi}$ results in a resolvent with an unsatisfiable condition. The first Backtrack-step is possible since $\varphi_{\mathsf{r}}|_{\psi_1}$ and φ_1^+ are inactive, as they require $X_1 < 5000$ for the first argument X_1 of their body-literal, but φ_2^+ ensures $Y_1 > 5000$ for the first argument Y_1 of its head-literal. Moreover, $\varphi_{\mathsf{r}}|_{\psi_2}$ and φ_2^+ are blocked, as $\varphi_{\mathsf{r}}|_{\psi_2} \sqsubseteq \varphi_2^+$. The second Backtrack-step is performed since $\varphi_{\mathsf{r}}|_{\psi_1}$, $\varphi_{\mathsf{r}}|_{\psi_2}$, φ_1^+, and φ_2^+ are blocked (as $\varphi_{\mathsf{r}}|_{\psi_1} \sqsubseteq \varphi_1^+$ and $\varphi_{\mathsf{r}}|_{\psi_2} \sqsubseteq \varphi_2^+$). The third Backtrack-step is possible since $\varphi_{\mathsf{r}}|_{\psi_1}$ and φ_1^+ are blocked, and $\varphi_{\mathsf{r}}|_{\psi_2}$ and φ_2^+ cannot be applied without applying φ_1^+ first, so they are inactive. Thus, we reach a state where the only fact φ_{f} is blocked and hence Prove applies.

To see an example for Covered, assume that we apply $\varphi_{\mathsf{r}}|_{\psi_1}$ *twice* before using Accelerate. Then the following derivation yields the trace that we obtained after the first acceleration step above:

$$\begin{aligned}
&(\Phi, [\varphi_{\mathsf{f}}, \varphi_{\mathsf{r}}|_{\psi_1}], [\varnothing, \varnothing, \varnothing]) \\
\overset{\mathrm{S}}{\rightsquigarrow}\ &(\Phi, [\varphi_{\mathsf{f}}, \varphi_{\mathsf{r}}|_{\psi_1}, \varphi_{\mathsf{r}}|_{\psi_1}], [\varnothing, \varnothing, \varnothing, \varnothing]) \\
\overset{\mathrm{A}}{\rightsquigarrow}\ &(\Pi_1, [\varphi_{\mathsf{f}}, \varphi_{\mathsf{r}}|_{\psi_1}, \varphi_1^+], [\varnothing, \varnothing, \varnothing, \{\varphi_1^+\}]) \\
\overset{\mathrm{C}}{\rightsquigarrow}\ &(\Pi_1, [\varphi_{\mathsf{f}}, \varphi_{\mathsf{r}}|_{\psi_1}], [\varnothing, \varnothing, \{\varphi_1^+\}]) && (\text{as } [\varphi_{\mathsf{r}}|_{\psi_1}, \varphi_1^+] \sqsubset \varphi_1^+) \\
\overset{\mathrm{C}}{\rightsquigarrow}\ &(\Pi_1, [\varphi_{\mathsf{f}}], [\varnothing, \{\varphi_{\mathsf{r}}|_{\psi_1}\}]) && (\text{as } \varphi_{\mathsf{r}}|_{\psi_1} \sqsubset \varphi_1^+) \\
\overset{\mathrm{S}}{\rightsquigarrow}\ &(\Pi_1, [\varphi_{\mathsf{f}}, \varphi_1^+], [\varnothing, \{\varphi_{\mathsf{r}}|_{\psi_1}\}, \varnothing]) && (\dagger)
\end{aligned}$$

As one can see in the example above, our calculus uses *forward reasoning*, i.e., it starts with a fact and resolves it with rules until a query applies. Alternatively,

one could use *backward reasoning* by starting with a query and resolving it with rules until a fact applies, as in logic programming.

Our calculus could easily be adapted for backward reasoning. Then it would start resolving with a query and aim for resolving with a fact, while all other aspects of the calculus would remain unchanged. Such an adaption would be motivated by examples like

$$\begin{aligned} \mathsf{F}(\ldots) \wedge \ldots &\implies \mathsf{G}(\ldots) \\ \mathsf{G}(\ldots) \wedge \ldots &\implies \mathsf{H}(\ldots) \\ \mathsf{G}(\ldots) \wedge \ldots &\implies \bot \end{aligned}$$

where H is the entry-point of a satisfiable sub-problem. With forward reasoning, ADCL might spend lots of time on that sub-problem, whereas unsatisfiability would be proven after just two steps with backward reasoning. However, in our tests, backward reasoning did not help on any example. Presumably, the reason is that the benchmark set from our evaluation does not contain examples with such a structure. Thus, we did not pursue this approach any further.

3.3 Properties of ADCL

In this section, we investigate the main properties of ADCL. Most importantly, ADCL is sound.

Theorem 12 (Soundness). *If $\Phi \rightsquigarrow^* \mathsf{sat}$, then Φ is satisfiable. If $\Phi \rightsquigarrow^* \mathsf{unsat}$, then Φ is unsatisfiable.*

Proof (Sketch).[3] For unsat, we have $\Phi \rightsquigarrow^* (\Pi, \vec{\varphi}, \vec{B}) \rightsquigarrow \mathsf{unsat}$ where $\Pi \equiv_{\mathcal{A}} \Phi$ and $\vec{\varphi} \in \mathsf{sip}(\Pi)^*$ is a refutation. For sat, assume that Φ is unsatisfiable, but $\Phi \rightsquigarrow s = (\Phi, [], [\varnothing]) \rightsquigarrow^* (\Pi, [], [B]) = s' \rightsquigarrow \mathsf{sat}$. Then there is a refutation $\vec{\varphi} \in \mathsf{sip}(\Pi)^*$ that is minimal in the sense that $\varphi_i \not\sqsubseteq \mathsf{sip}(\Pi)$ for all $1 \leq i \leq |\vec{\varphi}|$ and $\vec{\varphi}' \not\sqsubseteq \mathsf{sip}(\Pi)$ for all infixes $\vec{\varphi}'$ of $\vec{\varphi}$ whose length is at least 2. We say that $\vec{\varphi}$ is *disabled* by a state $(\Pi', \vec{\varphi}', \vec{B}')$ if $\vec{\varphi}'$ has a prefix $[\varphi'_i]_{i=1}^k$ such that $\varphi_i \equiv_{\mathcal{A}} \varphi'_i$ for all $1 \leq i \leq k$ and $\varphi_{k+1} \equiv_{\mathcal{A}} \varphi$ for some $\varphi \in B'_k$. Then $\vec{\varphi}$ is disabled by s', but not by s. Let $s^{(i)}$ be the last state in the derivation $\Phi \rightsquigarrow^* \mathsf{sat}$ where $\vec{\varphi}$ is enabled. Then $s^{(i)} \overset{\mathrm{S}}{\rightsquigarrow} s^{(i+1)}$ would imply that $\vec{\varphi}$ is enabled in $s^{(i+1)}$; $s^{(i)} \overset{\mathrm{A}}{\rightsquigarrow} s^{(i+1)}$ would imply that two consecutive clauses in $\vec{\varphi}$ are both equivalent to the newly learned clause, contradicting minimality of $\vec{\varphi}$; $s^{(i)} \overset{\mathrm{C}}{\rightsquigarrow} s^{(i+1)}$ would imply that the trace of $s^{(i)}$ is not minimal, which also contradicts minimality of $\vec{\varphi}$; and $s^{(i)} \overset{\mathrm{B}}{\rightsquigarrow} s^{(i+1)}$ would imply that an element of $\vec{\varphi}$ is strictly redundant w.r.t. the last set of blocking clauses in $s^{(i)}$, which again contradicts minimality of $\vec{\varphi}$. Hence, we derived a contradiction. □

Another important property of our calculus is that it cannot get "stuck" in states other than sat or unsat.

[3] All full proofs can be found in the extended version [26].

Theorem 13 (Normal Forms). *If $\Phi \rightsquigarrow^+ s$ where s is in normal form w.r.t. $\rightsquigarrow$, then $s \in \{\mathsf{sat}, \mathsf{unsat}\}$.*

Clearly, our calculus admits many unintended derivations, e.g., by applying Step over and over again with recursive CHCs instead of accelerating them. To prevent such derivations, a *reasonable strategy* is required.

Definition 14 (Reasonable Strategy). *We call a strategy for $\rightsquigarrow$* reasonable *if the following holds:*

(1) If $(\Pi, \vec{\varphi}, \vec{B}) \rightsquigarrow^+ (\Pi, \vec{\varphi} :: \vec{\varphi}', \vec{B}')$ for some $\vec{\varphi}'$ as in the definition of Covered, then Covered is used.
(2) Accelerate is used with higher preference than Step.
(3) Accelerate is only applied to the shortest recursive suffix $\vec{\varphi}^{\circlearrowleft}$ such that $\mathsf{accel}(\vec{\varphi}^{\circlearrowleft})$ is not redundant w.r.t. $\mathsf{sip}(\Pi)$.
(4) If $\vec{\varphi} = []$, then Step is only applied with facts or conditional empty clauses.

We write $\rightsquigarrow_{rs}$ for the relation that results from $\rightsquigarrow$ by imposing a reasonable strategy.

Def. 14 (1) ensures that we backtrack if we added a redundant sequence $\vec{\varphi}'$ of CHCs to the trace. However, for refutational completeness (Thm. 15), it is important that the application of Covered is only enforced if no new clauses have been learned while constructing $\vec{\varphi}'$ (i.e., Π remains unchanged in the derivation $(\Pi, \vec{\varphi}, \vec{B}) \rightsquigarrow^+ (\Pi, \vec{\varphi} :: \vec{\varphi}', \vec{B}')$). The reason is that after applying Accelerate, the trace might have the form $\vec{\varphi} = \vec{\varphi}_1 :: \vec{\varphi}_2 :: \mathsf{accel}(\varphi^{\circlearrowleft})$ where $\vec{\varphi}_2 :: \mathsf{accel}(\varphi^{\circlearrowleft}) \sqsubset \mathsf{accel}(\varphi^{\circlearrowleft})$ even if $\vec{\varphi}_2 :: \varphi^{\circlearrowleft}$ was non-redundant before learning $\mathsf{accel}(\varphi^{\circlearrowleft})$. If we enforced backtracking via Covered in such situations (which would yield the trace $\vec{\varphi}_1 :: \vec{\varphi}_2$), then to maintain refutational completeness, we would have to ensure that we eventually reach a state with the trace $\vec{\varphi}_1 :: \mathsf{accel}(\varphi^{\circlearrowleft}) \sqsubseteq \vec{\varphi}$. However, this cannot be guaranteed, since our calculus does not terminate in general (see Thm. 18).

Def. 14 (2) ensures that we do not "unroll" recursive derivations more than once via Step, but learn new clauses that cover arbitrarily many unrollings via Accelerate instead.

Def. 14 (3) has two purposes: First, it prevents us from learning redundant clauses, as we must not apply Accelerate if $\mathsf{accel}(\vec{\varphi}^{\circlearrowleft})$ is redundant. Second, it ensures that we accelerate "short" recursive suffixes first. The reason is that if $\vec{\varphi} = \vec{\varphi}_1 :: \vec{\varphi}_2 :: \vec{\varphi}_3$ where $\vec{\varphi}_2 :: \vec{\varphi}_3$ and $\vec{\varphi}_3$ are recursive, then

$$\mathsf{grnd}(\mathsf{accel}(\vec{\varphi}_2 :: \vec{\varphi}_3)) \overset{\text{Def. 4}}{=} \bigcup_{n \in \mathbb{N}_{\geq 1}} \mathsf{grnd}((\vec{\varphi}_2 :: \vec{\varphi}_3)^n)$$
$$\subseteq \bigcup_{n \in \mathbb{N}_{\geq 1}} \bigcup_{m \in \mathbb{N}_{\geq 1}} \mathsf{grnd}((\vec{\varphi}_2 :: \vec{\varphi}_3^m)^n) \overset{\text{Def. 4}}{=} \mathsf{grnd}(\mathsf{accel}(\vec{\varphi}_2 :: \mathsf{accel}(\vec{\varphi}_3))),$$

but the other direction ("$\supseteq$") does not hold in general. So in this way, we learn more general clauses.

Def. 14 (4) ensures that the first element of $\vec{\varphi}$ is always a fact or a conditional empty clause. For unsatisfiable CHC problems, the reason is that Refute will

never apply if $\vec{\varphi}$ starts with a rule or a query. For satisfiable CHC problems, Prove only applies if all facts and conditional empty clauses are blocked. But in order to block them eventually, we have to add them to the trace via Step, which is only possible if $\vec{\varphi}$ is empty.

Despite the restrictions in Def. 14, our calculus is still refutationally complete.

Theorem 15 (Refutational Completeness). *If Φ is unsatisfiable, then*

$$\Phi \rightsquigarrow_{rs}^{*} \mathsf{unsat}.$$

Proof (Sketch). Given a refutation $\vec{\varphi}$, one can inductively define a derivation $\Phi \rightsquigarrow_{rs}^{*} \mathsf{unsat}$ where each step applies Accelerate or Step. For the latter, it is crucial to choose the next clause in such a way that it corresponds to as many steps from $\vec{\varphi}$ as possible, and that it is maximal w.r.t. $\sqsubset$, to avoid the necessity to backtrack via Covered. □

However, in general our calculus does not terminate, even with a reasonable strategy. Note that even though CHC-SAT is undecidable for, e.g., CHCs over the theory LIA, non-termination of $\rightsquigarrow_{rs}$ is not implied by soundness of ADCL. The reason is that we assume oracles for undecidable sub-problems like SMT, checking redundancy, and acceleration. As acceleration may introduce non-linear integer arithmetic, both SMT and checking redundancy may even become undecidable when analyzing CHCs over a decidable theory like LIA.

To prove non-termination, we extend our calculus by one additional component: A mapping $\mathcal{L} : \mathsf{sip}(\Pi) \to \mathcal{P}(\mathsf{sip}(\Phi)^*)$ from $\mathsf{sip}(\Pi)$ to regular languages over $\mathsf{sip}(\Phi)$, where $\mathcal{P}(\mathsf{sip}(\Phi)^*)$ denotes the power set of $\mathsf{sip}(\Phi)^*$. We will show that this mapping gives rise to an alternative characterization of the ground instances of $\mathsf{sip}(\Pi)$, which will be exploited in our non-termination proof (Thm. 18). Moreover, this mapping is also used in our implementation to check redundancy, see Sect. 4. To extend our calculus, we lift $\mathcal{L}$ from $\mathsf{sip}(\Pi)$ to $\mathsf{sip}(\Pi)^*$ as follows:

$$\mathcal{L}(\varepsilon) := \varepsilon \qquad\qquad \mathcal{L}(\vec{\pi} :: \pi) := \mathcal{L}(\vec{\pi}) :: \mathcal{L}(\pi)$$

Here, "::" is also used to denote language concatenation, i.e., we have

$$\mathcal{L}_1 :: \mathcal{L}_2 := \{\vec{\pi}_1 :: \vec{\pi}_2 \mid \vec{\pi}_1 \in \mathcal{L}_1, \vec{\pi}_2 \in \mathcal{L}_2\}.$$

So while we lift other notations to sequences of transitions via resolution, $\mathcal{L}(\vec{\tau})$ does *not* stand for $\mathcal{L}(\mathsf{res}(\vec{\tau}))$.

Definition 16 (ADCL with Regular Languages). *We extend states (see Def. 9) by a fourth component $\mathcal{L} : \mathsf{sip}(\Pi) \to \mathcal{P}(\mathsf{sip}(\Phi)^*)$. The rules* Init *and* Accelerate *of the ADCL calculus (see Def. 10) are adapted as follows:*

$$\frac{\mathcal{L}(\varphi) = \{\varphi\} \text{ for all } \varphi \in \mathsf{sip}(\Phi)}{\Phi \rightsquigarrow_{rs} (\Phi, [], [\varnothing], \mathcal{L})} \text{ (Init)}$$

$$\frac{\vec{\varphi}^{\circlearrowleft} \text{ is recursive} \quad |\vec{\varphi}^{\circlearrowleft}| = |\vec{B}^{\circlearrowleft}| \quad \mathsf{accel}(\vec{\varphi}^{\circlearrowleft}) = \varphi \quad \mathcal{L}' = \mathcal{L} \uplus (\varphi \mapsto \mathcal{L}(\vec{\varphi}^{\circlearrowleft})^{+})}{(\Pi, \vec{\varphi} :: \vec{\varphi}^{\circlearrowleft}, \vec{B} :: \vec{B}^{\circlearrowleft}, \mathcal{L}) \rightsquigarrow_{rs} (\Pi \cup \{\varphi\}, \vec{\varphi} :: \varphi, \vec{B} :: \{\varphi\}, \mathcal{L}')} \text{ (Accelerate)}$$

All other rules from Def. 10 leave the last component of the state unchanged.

Here, $\mathcal{L}(\pi)^+$ denotes the "Kleene plus" of $\mathcal{L}(\pi)$, i.e., we have

$$\mathcal{L}(\pi)^+ := \bigcup_{n \in \mathbb{N}_{\geq 1}} \mathcal{L}(\pi)^n.$$

Note that Def. 16 assumes a reasonable strategy (indicated by the notation $\rightsquigarrow_{rs}$). Hence, when Accelerate is applied, we may assume $\varphi \notin \mathsf{sip}(\Pi) = \mathrm{dom}(\mathcal{L})$. Otherwise, φ would be redundant and hence a reasonable strategy would not allow the application of Accelerate. For this reason, we may write "$\uplus$" in the definition of $\mathcal{L}'$.

The following lemma allows us to characterize the ground instances of elements of $\mathsf{sip}(\Pi)$ via $\mathcal{L}$. Here, we lift grnd to sets by defining $\mathsf{grnd}(X) := \bigcup_{x \in X} \mathsf{grnd}(x)$, where X may be a set of CHCs or a language over CHCs. Thus, $\mathsf{grnd}(\mathcal{L}(\pi))$ is the set of all ground instances of the final resolvents of the sequences in $\mathcal{L}(\pi)$.

Lemma 17. *If* $\Phi \rightsquigarrow_{rs}^* (\Pi, \vec{\varphi}, \vec{B}, \mathcal{L})$ *and* $\pi \in \mathsf{sip}(\Pi)$*, then* $\mathsf{grnd}(\pi) = \mathsf{grnd}(\mathcal{L}(\pi))$.

Now we are ready to prove that, even with a reasonable strategy, ADCL does not terminate.

Theorem 18 (Non-Termination). *There exists a satisfiable CHC problem* Φ *such that* $\Phi \not\rightsquigarrow_{rs}^* \mathsf{sat}$*. Thus,* $\rightsquigarrow_{rs}$ *does not terminate.*

Proof (Sketch). One can construct a satisfiable CHC problem Φ such that all (infinitely many) resolution sequences with Φ are *square-free*, i.e., they do not contain a non-empty subsequence of the form $\vec{\varphi} :: \vec{\varphi}$. For example, this can be achieved by encoding the differences between subsequent numbers of the Thue-Morse sequence [45,46]. As an invariant of our calculus, $\mathcal{L}(\Pi)$ just contains finitely many square-free words for any reachable state $(\Pi, \vec{\varphi}, \vec{B}, \mathcal{L})$. As $\mathsf{grnd}(\Pi) = \mathsf{grnd}(\mathcal{L}(\Pi))$, this means that Π cannot cover all resolution sequences with Φ. Thus, the assumption $\Phi \rightsquigarrow_{rs}^* \mathsf{sat}$ results in a contradiction. □

The construction from the proof of Thm. 18 can also be used to show that there are non-terminating derivations $\Phi \rightsquigarrow_{rs} s_1 \rightsquigarrow_{rs} s_2 \rightsquigarrow_{rs} \dots$ where Φ is unsatisfiable. However, in this case there is also another derivation $\Phi \rightsquigarrow_{rs}^* \mathsf{unsat}$ due to refutational completeness (see Thm. 15).

4 Implementing ADCL

We now explain how we implemented ADCL efficiently in our tool LoAT. Here we focus on proving unsatisfiability. The reason is that our implementation cannot prove sat at the moment, since it uses certain approximations that are incorrect for sat, as detailed below. Thus, when applying Prove, our implementation returns unknown instead of sat. Our implementation uses Yices [18] and Z3 [44] for SMT solving. Moreover, it is based on the acceleration technique from [23], whose implementation solves recurrence relations with PURRS [4].

Checking Redundancy To check redundancy in Accelerate (as required for reasonable strategies in Def. 14), we use the fourth component $\mathcal{L}$ of states introduced in Def. 16. More precisely, for Accelerate, we check if $\mathcal{L}(\vec{\varphi}^{\circlearrowleft})^+ \subseteq \mathcal{L}(\varphi)$ holds for some learned clause φ. In that case, $\mathsf{accel}(\vec{\varphi}^{\circlearrowleft})$ is redundant due to Lemma 17. Since $\mathcal{L}(\vec{\varphi}^{\circlearrowleft})^+ \subseteq \mathcal{L}(\varphi)$ is simply an inclusion check for regular languages, it can be implemented efficiently using finite automata. Our implementation uses the automata library libFAUDES [41].

However, this is just a sufficient criterion for redundancy. For example, a learned clause might be redundant w.r.t. an original clause, but such redundancies cannot be detected using $\mathcal{L}$. To see this, note that we have $|\mathcal{L}(\varphi)| = 1$ if φ is an original clause, but $|\mathcal{L}(\varphi)| = \infty$ if φ is a learned clause.

For Covered, we also check redundancy via $\mathcal{L}$, but if $\vec{\varphi}' = \varphi'$, i.e., if $|\vec{\varphi}'| = 1$, then we only apply Covered if φ' is an original clause. Then $\mathcal{L}(\varphi') \subseteq \mathcal{L}(\varphi)$ for some $\varphi \neq \varphi'$ implies that φ is a learned clause. Hence, we have $\mathcal{L}(\varphi') \subset \mathcal{L}(\varphi)$, as $|\mathcal{L}(\varphi')| = 1 < |\mathcal{L}(\varphi)| = \infty$. This is just a heuristic, as even $\mathcal{L}(\varphi') \subset \mathcal{L}(\varphi)$ just implies $\varphi' \sqsubseteq \varphi$, but not $\varphi' \sqsubset \varphi$. To see this, consider an original clause $\varphi = (\mathsf{F}(X) \implies \mathsf{F}(0))$. Then $\mathcal{L}(\varphi) = \{\varphi\}$, $\mathsf{accel}(\varphi) \equiv_{\mathcal{A}} \varphi$ (but not necessarily $\mathsf{accel}(\varphi) = \varphi$, as $\mathsf{accel}(\varphi)$ and φ might differ syntactically), and $\mathcal{L}(\mathsf{accel}(\varphi)) = \mathcal{L}(\varphi)^+$. So we have $\mathcal{L}(\varphi) \subset \mathcal{L}(\mathsf{accel}(\varphi))$ and $\varphi \sqsubseteq \mathsf{accel}(\varphi)$, but $\varphi \not\sqsubset \mathsf{accel}(\varphi)$. This is uncritical for proving unsat, but a potential soundness issue for proving sat, which is one reason why our current implementation cannot prove sat.

Implementing Step and Blocked Clauses To find an active clause in Step, we proceed as described before Def. 8, i.e., we search for a suitable element of $\mathsf{sip}(\Pi)$ "on the fly". So we search for a clause $\varphi \in \Pi$ whose body-literal unifies with the head-literal of $\mathsf{res}(\vec{\varphi})$ using an mgu θ. Then we use an SMT solver to check whether

$$\theta(\mathsf{cond}(\mathsf{res}(\vec{\varphi}))) \wedge \theta(\mathsf{cond}(\varphi)) \wedge \bigwedge_{\pi \in B \cap \mathsf{sip}(\varphi)} \neg\theta(\mathsf{cond}(\pi)) \qquad \text{(Step–SMT)}$$

is satisfiable, where B is the last element of $\vec{B}$. Here, we assume that $\mathsf{res}(\vec{\varphi})$ and φ are variable disjoint (and thus the mgu θ exists). If we find a model σ for Step–SMT, then we apply Step with $\varphi|_{\mathsf{sip}(\mathsf{cond}(\varphi),\sigma)}$. So to exclude blocked clauses, we do not use the redundancy check based on $\mathcal{L}$ explained above, but we conjoin the negated conditions of certain blocked clauses to Step–SMT. To see why we only consider blocked clauses from $\mathsf{sip}(\varphi)$, consider the case that $B = \{\pi\}$ is a singleton. Note that both $\theta(\mathsf{cond}(\varphi))$ and $\theta(\mathsf{cond}(\pi))$ might contain variables that do not occur as arguments of predicates in the (unified) head- or body-literals. So if

$$\begin{aligned}
\varphi &\equiv_{\mathcal{A}} \forall \vec{X}, \vec{Y}_{\varphi}, \vec{X}'.\ \mathsf{F}(\vec{X}) \wedge \psi_{\varphi}(\vec{X}, \vec{Y}_{\varphi}, \vec{X}') \implies \mathsf{G}(\vec{X}'),\\
\varphi' &\equiv_{\mathcal{A}} \forall \vec{X}, \vec{Y}_{\varphi'}, \vec{X}'.\ \mathsf{F}(\vec{X}) \wedge \psi_{\varphi'}(\vec{X}, \vec{Y}_{\varphi'}, \vec{X}') \implies \mathsf{G}(\vec{X}'), \qquad \text{and}\\
\pi &\equiv_{\mathcal{A}} \forall \vec{X}, \vec{Y}_{\pi}, \vec{X}'.\ \mathsf{F}(\vec{X}) \wedge \psi_{\pi}(\vec{X}, \vec{Y}_{\pi}, \vec{X}') \implies \mathsf{G}(\vec{X}'),
\end{aligned}$$

for some $\varphi' \in \mathsf{sip}(\varphi)$, then $\varphi' \sqsubseteq \pi$ iff

$$\models_{\mathcal{A}} \psi_{\varphi'} \implies \exists \vec{Y}_\pi.\ \psi_\pi. \qquad (\sqsubseteq\text{–EQUIV})$$

Thus, to ensure that we only find models σ such that $\mathsf{sip}(\mathsf{cond}(\varphi), \sigma)$ is not blocked by π, we would have to conjoin

$$\neg(\psi_\varphi \implies \exists \vec{Y}_\pi.\ \psi_\pi) \quad \equiv_{\mathcal{A}} \quad \psi_\varphi \wedge \forall \vec{Y}_\pi.\ \neg\psi_\pi$$

to the SMT problem. Unfortunately, as SMT solvers have limited support for quantifiers, such an encoding is impractical. Hence, we again use a sufficient criterion for redundancy: If

$$\models_{\mathcal{A}} \psi_{\varphi'} \implies \psi_\pi, \qquad (\sqsubseteq\text{–SUFFICIENT})$$

then $\sqsubseteq$–EQUIV trivially holds as well. So to exclude conjunctive variants φ' of φ where $\sqsubseteq$–SUFFICIENT is valid, we add

$$\neg(\psi_\varphi \implies \psi_\pi) \quad \equiv_{\mathcal{A}} \quad \psi_\varphi \wedge \neg\psi_\pi \qquad (\not\sqsubseteq\text{–SUFFICIENT})$$

to the SMT problem. If $\vec{Y}_\pi \not\subseteq \vec{Y}_\varphi$, then satisfiability of $\not\sqsubseteq$–SUFFICIENT is usually trivial. Thus, to avoid increasing the size of the SMT problem unnecessarily, we only add $\not\sqsubseteq$–SUFFICIENT to the SMT problem if $\pi \in \mathsf{sip}(\varphi)$. Instead, we could try to rename variables from $\vec{Y}_\pi$ to enforce $\vec{Y}_\pi \subseteq \vec{Y}_\varphi$. However, it is difficult to predict which renaming is the "right" one, i.e., which renaming would allow us to prove redundancy.

If $B \cap \mathsf{sip}(\varphi)$ contains several clauses $\pi_1, \ldots, \pi_\ell$, then $\sqsubseteq$–SUFFICIENT becomes

$$\models_{\mathcal{A}} \psi_{\varphi'} \implies \mathsf{cond}(\pi_1) \quad \text{or} \quad \ldots \quad \text{or} \quad \models_{\mathcal{A}} \psi_{\varphi'} \implies \mathsf{cond}(\pi_\ell) \quad (\sqsubseteq\text{–SUFFICIENT}^+)$$

Instead, our encoding excludes syntactic implicants φ' of φ where

$$\models_{\mathcal{A}} \psi_{\varphi'} \implies \mathsf{cond}(\pi_1) \vee \ldots \vee \mathsf{cond}(\pi_\ell) \qquad (\sqsubseteq\text{–INSUFFICIENT}^+)$$

which is a necessary, but not a sufficient condition for $\sqsubseteq$–SUFFICIENT$^+$. To see why this is not a problem, first note that $\sqsubseteq$–SUFFICIENT$^+$ trivially holds if $\psi_{\varphi'} \in \{\mathsf{cond}(\pi_i) \mid 1 \leq i \leq \ell\}$. Otherwise, we have

$$\models_{\mathcal{A}} (\mathsf{cond}(\pi_1) \vee \ldots \vee \mathsf{cond}(\pi_\ell)) \implies \bigvee \mathsf{sip}(\psi_\varphi) \setminus \{\psi_{\varphi'}\}$$

because we assumed $\psi_{\varphi'} \notin \{\mathsf{cond}(\pi_i) \mid 1 \leq i \leq \ell\}$, which implies $\{\mathsf{cond}(\pi_i) \mid 1 \leq i \leq \ell\} \subseteq \mathsf{sip}(\psi_\varphi) \setminus \{\psi_{\varphi'}\}$. Together with $\sqsubseteq$–INSUFFICIENT$^+$, this implies

$$\models_{\mathcal{A}} \psi_{\varphi'} \implies \bigvee \mathsf{sip}(\psi_\varphi) \setminus \{\psi_{\varphi'}\}.$$

Therefore, we have $\psi_\varphi \equiv_{\mathcal{A}} \bigvee \mathsf{sip}(\psi_\varphi) \setminus \{\psi_{\varphi'}\}$. Thus, we may assume that $\sqsubseteq$–INSUFFICIENT$^+$ implies redundancy without loss of generality. The reason

is that we could analyze the following equivalent CHC problem instead of Π, otherwise:

$$(\Pi \setminus \{\varphi\}) \cup (\mathsf{sip}(\varphi) \setminus \{\varphi'\})$$

Hence, in Step–SMT, we add (a variable-renamed variant of)

$$\begin{aligned}\neg(\psi_\varphi \implies \mathsf{cond}(\pi_1) \vee \ldots \vee \mathsf{cond}(\pi_\ell)) \quad &\equiv_{\mathcal{A}} \quad \psi_\varphi \wedge \neg\mathsf{cond}(\pi_1) \wedge \ldots \wedge \neg\mathsf{cond}(\pi_\ell) \\ &\equiv_{\mathcal{A}} \quad \mathsf{cond}(\varphi) \wedge \bigwedge_{\pi \in B \cap \mathsf{sip}(\varphi)} \neg\mathsf{cond}(\pi)\end{aligned}$$

to the SMT problem.

Example 19. Consider the state (†) from Ex. 11. First applying Step with $\varphi_\mathsf{r}|_{\psi_1}$ and then applying Covered yields

$$(\Pi_1, \vec{\varphi}, [\varnothing, \{\varphi_\mathsf{r}|_{\psi_1}\}, \{\varphi_\mathsf{r}|_{\psi_1}\}])$$

where $\vec{\varphi} = [\varphi_\mathsf{f}, \varphi_1^+]$. When attempting a Step with an element of $\mathsf{sip}(\varphi_\mathsf{r})$, we get:

$$\begin{aligned}\theta(\mathsf{cond}(\vec{\varphi})) &\equiv_{\mathcal{A}} X_1 = 0 \wedge X_2 = 5k \wedge N > 0 \wedge X_1' < 5001 \wedge X_1' = X_1 + N \wedge X_2' = X_2 \\ \theta(\mathsf{cond}(\varphi_\mathsf{r})) &\equiv_{\mathcal{A}} ((X_1' < 5k \wedge Y_2 = X_2') \vee (X_1' \geq 5k \wedge Y_2 = X_2' + 1)) \\ \bigwedge_{\pi \in B \cap \mathsf{sip}(\varphi_r)} \neg\theta(\mathsf{cond}(\pi)) = \neg\theta(\mathsf{cond}(\varphi_\mathsf{r}|_{\psi_1})) &\equiv_{\mathcal{A}} X_1' \geq 5k \vee Y_2 \neq X_2'\end{aligned}$$

Here, $5k$ abbreviates 5000. Then Step–SMT is equivalent to

$$X_1 = 0 \wedge X_2 = N = X_1' = X_2' = 5k \wedge Y_2 = 5001.$$

Hence, we have $\sigma \models_{\mathcal{A}} X_1' \geq 5k \wedge Y_2 = X_2' + 1$ for the unique model σ of Step–SMT, i.e., σ satisfies the second disjunct of $\mathsf{cond}(\varphi_\mathsf{r})$. Thus, we add $\varphi_\mathsf{r}|_{\mathsf{sip}(\mathsf{cond}(\varphi_r),\sigma)} = \varphi_\mathsf{r}|_{\psi_2}$ to the trace.

Leveraging Incremental SMT The search for suitable models can naturally be implemented via incremental SMT solving: When trying to apply Step, we construct θ in such a way that $\theta(\mathsf{cond}(\mathsf{res}(\vec{\varphi}))) = \mathsf{cond}(\mathsf{res}(\vec{\varphi}))$. This is easily possible, as θ just needs to unify predicates whose arguments are duplicate free and pairwise disjoint vectors of variables. Then we push

$$\theta(\mathsf{cond}(\varphi)) \wedge \bigwedge_{\pi \in B \cap \mathsf{sip}(\varphi)} \neg\theta(\mathsf{cond}(\pi)) \qquad \text{(Incremental)}$$

to the SMT solver. If the model from the previous resolution step can be extended to satisfy Incremental, then the SMT solver can do so, otherwise it searches for another model. If it fails to find a model, we pop Incremental, i.e., we remove it from the current SMT problem. Accelerate can be implemented similarly by popping $\theta(\mathsf{cond}(\vec{\varphi}^\circlearrowleft))$ and pushing $\theta(\mathsf{cond}(\varphi))$ instead.

Note that satisfiability of $\vec{\varphi}$ is an invariant of ADCL. Hence, as soon as the last element of $\vec{\varphi}$ is a query, Refute can be applied without further SMT checks. Otherwise, if Step cannot be applied with any clause, then Backtrack or Prove can be applied without further SMT queries.

Dealing with Incompleteness As mentioned in Sect. 3, we assumed that we have oracles for checking redundancy, satisfiability of $\mathsf{QF}(\mathcal{A})$-formulas, and acceleration when we formalized ADCL. As this is not the case in practice, we now explain how to proceed if those techniques fail or approximate.

As explained above, SMT is needed for checking activity in Step. If the SMT solver fails, we assume inactivity. Thus, we do not exhaust the entire search space if we falsely classify active clauses as inactive. Hence, we may miss refutations, which is another reason why our current implementation cannot prove sat.

Regarding acceleration, our implementation of accel may return under-approximations, i.e., we just have $\mathsf{grnd}(\mathsf{accel}(\varphi)) \subseteq \bigcup_{n \in \mathbb{N}_{\geq 1}} \mathsf{grnd}(\varphi^n)$. While this is uncritical for correctness by itself (as learned clauses are still entailed by Φ), it weakens our heuristic for redundancy via $\mathcal{L}$, as we no longer have $\mathsf{grnd}(\varphi) = \mathsf{grnd}(\mathcal{L}(\varphi))$, but just $\mathsf{grnd}(\varphi) \subseteq \mathsf{grnd}(\mathcal{L}(\varphi))$ for learned clauses φ.

Another pitfall when using under-approximating acceleration techniques is that we may have $\vec{\varphi}^{\circlearrowleft} \not\sqsubseteq \mathsf{accel}(\vec{\varphi}^{\circlearrowleft})$. In this case, applying Accelerate can result in an inconsistent trace where $\mathsf{cond}(\vec{\varphi})$ is unsatisfiable. To circumvent this problem, we only add $\mathsf{accel}(\vec{\varphi}^{\circlearrowleft})$ to the trace after removing $\vec{\varphi}^{\circlearrowleft}$ if doing so results in a consistent trace. Here, we could do better by taking the current model σ into account when accelerating $\vec{\varphi}^{\circlearrowleft}$ in order to ensure $\sigma \models_{\mathcal{A}} \mathsf{cond}(\mathsf{accel}(\vec{\varphi}^{\circlearrowleft}))$. We leave that to future work.

Restarts When testing our implementation, we noticed that several instances "jiggled", i.e., they were solved in some test runs, but failed in others. The reason is a phenomenon that is well-known in SAT solving, called "heavy-tail behavior". Here, the problem is that the solver sometimes gets "stuck" in a part of the state space whose exploration is very expensive, even though finding a solution in another part of the search space is well within the solver's capabilities. This problem also occurs in our implementation, due to the depth-first strategy of our solver (where derivations may even be non-terminating, see Thm. 18). To counter this problem, SAT solvers use restarts [28], where one of the most popular approaches has been proposed by Luby et al. [42]. For SAT solving, the idea is to restart the search after a certain number of conflicts, where the number of conflicts for the next restart is determined by the *Luby sequence*, scaled by a parameter u. When restarting, randomization is used to avoid revisiting the same part of the search space again. We use the same strategy with $u = 10$, where we count the number of learned clauses instead of the number of conflicts. To restart the search, we clear the trace, change the seed of the SMT solver (which may result in different models such that we may use different syntactic implicants), and shuffle the vectors of clauses (to change the order in which clauses are used for resolution).

5 Related Work and Experiments

We presented the novel ADCL calculus for (dis)proving satisfiability of CHCs. Its distinguishing feature is its use of acceleration for learning new clauses. For

unsatisfiability, these learned clauses often enable very short resolution proofs for CHC problems whose original clauses do not admit short resolution proofs. For satisfiability, learned clauses often allow for covering the entire (usually infinite) search space by just considering finitely many resolution sequences.

Related Work The most closely related work is [34], where acceleration is used in two ways: (1) as preprocessing and (2) to generalize interpolants in a CEGAR loop. In contrast to (1), we use acceleration "on the fly" to accelerate resolvents. In contrast to (2), we do not use abstractions, so our learned clauses can directly be used in resolution proofs. Moreover, [34] only applies acceleration to conjunctive clauses, whereas we accelerate conjunctive variants of arbitrary clauses. So in our approach, acceleration techniques are applicable more often, which is particularly useful for finding long counterexamples. However, our approach *solely* relies on acceleration to handle recursive CHCs, whereas [34] incorporates acceleration techniques into a CEGAR loop, which can also analyze recursive CHCs without accelerating them. Thus, the approach from [34] is orthogonal to ADCL. Both (1) and (2) are implemented in Eldarica, but according to its authors, (2) is just supported for transition systems, but not yet for CHCs. Hence, we only considered (1) in our evaluation (named Eld. Acc. below). Earlier, an approach similar to (2) has been proposed in [13], but to the best of our knowledge, it has never been implemented.

Transition power abstraction (TPA) [7] computes a sequence of over-approximations for transition systems where the n^{th} element captures 2^n instead of just n steps of the transition relation. So like ADCL, TPA can help to find long refutations quickly, but in contrast to ADCL, TPA relies on over-approximations.

Some leading techniques for CHC-SAT like GPDR [33] and, in particular, the Spacer algorithm [37], are adaptions of the IC3 algorithm [11] from transition systems to CHCs. IC3 computes a sequence of abstractions of reachable states, aiming to find an abstraction that is inductive w.r.t. the transition relation and implies safety.

Other approaches for CHC-SAT are based on interpolation [17,35], CEGAR and predicate abstraction [29, 35], automata [17], machine learning [20, 51], bounded model checking (BMC) [6], or combinations thereof.

Related approaches for transition systems include [5] and [10]. The approach of [5] uses acceleration to analyze a sequence of *flattenings* of a given transition system, i.e., under-approximations without nested loops, until a counterexample is found or a fixpoint is reached. Like ADCL, this approach does not terminate in general. However, it does terminate for so-called *flattable* systems. Whether ADCL terminates for flattable systems as well is an interesting question for future work. In contrast to ADCL, [5] has no notion of learning or redundancy, so that the same computations may have to be carried our several times for different flattenings.

The technique of [10] also lifts acceleration techniques to transition systems, but circumvents non-termination by using approximative acceleration techniques in the presence of disjunctions. In contrast, ADCL handles disjunctions via

syntactic implicants. Like ADCL, [10, Alg. 2] learns new transitions (Line 9), but only if they are non-redundant (Line 8). However, it applies acceleration to all syntactic self-loops, whereas ADCL explores the state space starting from facts, such that only reachable loops are accelerated. Note that the approach from [10] is very similar to the approach that has been used by earlier versions of LoAT for proving non-termination [24]. We recently showed in [25] that for the purpose of proving non-termination, ADCL is superior to LoAT's earlier approach.

Finally, [40] uses under-approximating acceleration techniques to enrich the control-flow graph of C programs in order to find "deep counterexamples", i.e., long refutations. In contrast to ADCL, [40] relies on external model checkers for finding counterexamples, and it has no notion of redundancy so that the model checker may explore "superfluous" paths that use original instead of accelerated edges of the control-flow graph.

Regarding acceleration, there are many results regarding classes of loops over integer variables where linear arithmetic suffices to express their transitive closure, i.e., they can be accelerated within a decidable theory. The most important such classes are Difference Bounds [16], Octagons [8], Finite Monoid Affine Relations [21], and Vector Addition Systems with States [31]. In an orthogonal line of research, monotonicity-based acceleration techniques have been developed [22, 23, 40]. While the latter provide fewer theoretical guarantees in terms of completeness and whether the result can be expressed in a decidable logic or not, they are not restricted to loops whose transitive closure is definable in linear arithmetic.

Regarding other theories, the technique from [31] for Vector Addition Systems with States has also been applied to systems over rationals [48]. Similarly, monotonicity-based approaches immediately carry over to rationals or reals. The only approach for acceleration in the presence of Boolean variables that we are aware of is [47]. However, this technique yields over-approximations.

Finally, some acceleration techniques for arrays have been proposed, e.g., [15, 32]. The approach of [15] improves the framework of [39] to reason about programs with arrays using a first-order theorem prover by integrating specialized techniques for dealing with array accesses where the indices are monotonically increasing or decreasing. The technique of [32] uses quantifier elimination techniques to accelerate loops where arrays can be separated into *read-* and *write-only* arrays.

Experiments So far, our implementation of ADCL in LoAT is restricted to integer arithmetic. Thus, to evaluate our approach, we used the examples from the category LIA-Lin (linear CHCs with linear integer arithmetic) from the CHC competition '22 [14], which contains numerous CHC problems resulting from actual program verification tasks. Somewhat surprisingly, these examples contain additional features like variables of type `Bool` and the operators `div` and `mod`. Since variables of type `Bool` are used in most of the examples, we extended our implementation with rudimentary support for `Bool`s. In particular, we implemented a simplistic acceleration technique for `Bool`s (note that we cannot use the

approach of [47], as it yields over-approximations). We excluded the 72 examples that use `div` or `mod`, as those operators are not supported by our implementation.

To accelerate CHCs where some variables are of type `Bool`, we use an adaption of the acceleration calculus from [23]. To apply it to $\varphi := (\mathsf{F}(\vec{X}) \wedge \psi \implies \mathsf{F}(\vec{Y}))$, φ needs to be *deterministic*, i.e., there must be a substitution θ such that $\psi \models_{\mathcal{A}} \vec{Y} = \theta(\vec{X})$ and $\mathcal{V}(\theta(\vec{X})) \subseteq \vec{X}$. Then LoAT has to compute a *closed form*, i.e., a vector $\vec{C}$ such that $\vec{C} \equiv_{\mathcal{A}} \theta^N(\vec{X})$. For integer variables, closed forms are computed via recurrence solving. For Boolean variables B, LoAT can only construct a closed form if there is a $k \in \mathbb{N}$ such that $\theta^k(B)$ does not contain Boolean variables, or $\theta^k(B) = \theta^{k+1}(B)$. Once a closed form has been computed, the calculus from [23] can be applied. However, in the presence of Booleans, it has to be restricted to theory-agnostic acceleration techniques. So more precisely, in the presence of Booleans, only the acceleration techniques *monotonic increase* and *monotonic decrease* from [23] can be used.

Using the remaining 427 examples, we compared our implementation with the leading CHC-SAT solvers Spacer [37] (which is part of Z3 [44]), Eldarica [35], and Golem [7]. Additionally, we compared with Z3's implementation of BMC. As mentioned above, Eldarica supports acceleration as preprocessing. Thus, besides Eldarica's default configuration (which does not use acceleration), we also compared with a configuration Eld. Acc. where we enabled this feature. By default, Golem uses the Spacer algorithm, but the Spacer implementation in Z3 performed better in our tests. Thus, we used Golem's implementation of TPA instead, which targets similar classes of examples like ADCL, as explained above. We used Z3 4.11.2, Eldarica 2.0.8, and Golem 0.3.0 and ran our experiments on StarExec [49] with a wallclock timeout of 300s, a cpu timeout of 1200s, and a memory limit of 128GB per example.

The results can be seen in Fig. 2. We evaluated all tools on the 209 examples that do not use `Bool`s (Int only) and on the entire benchmark set (Int & Bool). The table on the left shows that LoAT is very competitive w.r.t. proving unsat in terms of solved instances. The entries in the column "unique" show the number of examples where the respective tool succeeds and all others fail. Here we disregard Eld. Acc., as it would be pointless to consider several variants of the same algorithm in such a comparison. If we consider Eld. Acc. instead of Eldarica, then the numbers change according to the values given in parentheses.

The numbers indicate that LoAT is particularly powerful on examples that operate on `Int`s only, but it is also competitive for proving unsatisfiability of examples that may operate on `Bool`s, where it is only slightly weaker than Spacer and Z3 BMC. This is not surprising, as the core of LoAT's approach are its acceleration techniques, which have been designed for integers. In contrast, Spacer's algorithm is similar to GPDR [33], which generalizes the IC3 algorithm [11] from transition systems over Booleans to transition systems over theories (like integers), and BMC is theory agnostic.

The figure on the right shows how many proofs of unsatisfiability were found within a given runtime, by each tool. Here, all examples (Int & Bool) are taken into account. LoAT finds many proofs of unsatisfiability quickly (73 proofs within

		unsat		sat
		solved	unique	solved
Int only	LoAT	30	5 (5)	0
	Z3 BMC	24	1 (1)	23
	Spacer	24	0 (0)	93
	Eldarica	23	0 (–)	122
	Golem TPA	15	0 (0)	45
	Eld. Acc.	21	– (0)	105
Int & Bool	LoAT	78	11 (5)	0
	Z3 BMC	84	5 (5)	28
	Spacer	79	2 (2)	183
	Eldarica	53	0 (–)	165
	Golem TPA	56	0 (0)	73
	Eld. Acc.	56	– (4)	142

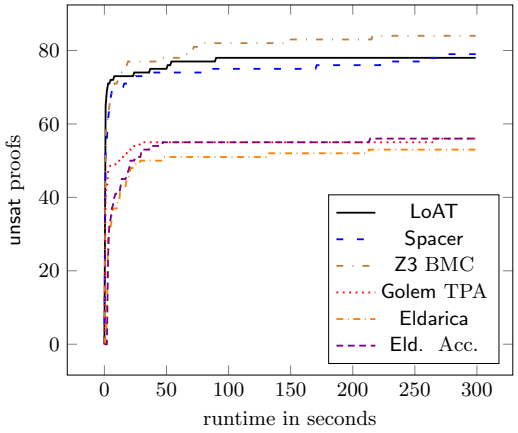


Fig. 2: Comparing LoAT with other CHC solvers

example	LoAT's refutation	original refutation
chc-LIA-Lin_043.smt2	6	965553
chc-LIA-Lin_045.smt2	2	684682683
chc-LIA-Lin_047.smt2	3	72536
chc-LIA-Lin_059.smt2	3	100000001
chc-LIA-Lin_154.smt2	2	134217729
chc-LIA-Lin_358.smt2	12	400005
chc-LIA-Lin_362.smt2	12	400005
chc-LIA-Lin_386.smt2	15	600003
chc-LIA-Lin_401.smt2	8	200005
chc-LIA-Lin_402.smt2	4	134217723
chc-LIA-Lin_405.smt2	9	100012

Table 1: Comparing lengths of refutation

8s). Z3 BMC catches up after 12s (73 proofs for both LoAT and Z3 BMC) and takes over the lead after 14s (LoAT 73, Z3 BMC 74). Spacer catches up with LoAT after 260s.

To illustrate LoAT's ability to find short refutations, Table 1 compares the number of resolution steps in LoAT's "accelerated" refutations (that also use learned clauses) with the corresponding refutations that only use original clauses. Here, we restrict ourselves to those instances that can only be solved by LoAT, as the unsatisfiable CHC problems that can also be solved by other tools usually already admit quite short refutations without learned clauses. To compute the length of the original refutations, we instrumented each predicate with an additional argument c. Moreover, we extended the condition of each fact $\psi \implies \mathsf{G}(..., c)$ with $c = 1$ and the condition of each rule $\mathsf{F}(..., c) \wedge \psi \implies \mathsf{G}(..., c')$ with $c' = c + 1$. Then the value of c before applying a query corresponds to the number of resolution steps that one would need if one only used original clauses, and it can

be extracted from the model found by the SMT solver. The numbers clearly show that learning clauses via acceleration allows to reduce the length of refutations dramatically. In 76 cases, LoAT learned clauses with non-linear arithmetic.

Our implementation is open-source and available on Github. For the sources, a pre-compiled binary, and more information on our evaluation, we refer to [1, 2]. In future work, we plan to extend our implementation to also prove sat, and we will investigate how to construct models for satisfiable CHC problems. Moreover, we want to add support for further theories by developing specialized acceleration techniques. Furthermore, we intend to lift ADCL to non-linear CHCs.

References

1. Artifact for "ADCL: Acceleration Driven Clause Learning for Constrained Horn Clauses" (2023). https://doi.org/10.5281/zenodo.8146788
2. Evaluation of "ADCL: Acceleration Driven Clause Learning for Constrained Horn Clauses" (2023), https://loat-developers.github.io/adcl-evaluation, source code of LoAT available at https://github.com/loat-developers/LoAT/tree/v0.4.0
3. Alt, L., Blicha, M., Hyvärinen, A.E.J., Sharygina, N.: SolCMC: Solidity compiler's model checker. In: CAV '22. pp. 325–338. LNCS 13371 (2022). https://doi.org/10.1007/978-3-031-13185-1_16
4. Bagnara, R., Pescetti, A., Zaccagnini, A., Zaffanella, E.: PURRS: Towards computer algebra support for fully automatic worst-case complexity analysis. CoRR **abs/cs/0512056** (2005). https://doi.org/10.48550/arXiv.cs/0512056
5. Bardin, S., Finkel, A., Leroux, J., Schnoebelen, P.: Flat acceleration in symbolic model checking. In: ATVA '05. pp. 474–488. LNCS 3707 (2005). https://doi.org/10.1007/11562948_35
6. Biere, A.: Bounded model checking. In: Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 739–764. IOS Press (2021). https://doi.org/10.3233/FAIA201002
7. Blicha, M., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: Transition power abstractions for deep counterexample detection. In: TACAS '22. pp. 524–542. LNCS 13243 (2022). https://doi.org/10.1007/978-3-030-99524-9_29
8. Bozga, M., Gîrlea, C., Iosif, R.: Iterating octagons. In: TACAS '09. pp. 337–351. LNCS 5505 (2009). https://doi.org/10.1007/978-3-642-00768-2_29
9. Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: CAV '10. pp. 227–242. LNCS 6174 (2010). https://doi.org/10.1007/978-3-642-14295-6_23
10. Bozga, M., Iosif, R., Konečný, F.: Relational analysis of integer programs. Tech. Rep. TR-2012-10, VERIMAG (2012), https://www-verimag.imag.fr/TR/TR-2012-10.pdf
11. Bradley, A.R.: SAT-based model checking without unrolling. In: VMCAI '11. pp. 70–87. LNCS 6538 (2011). https://doi.org/10.1007/978-3-642-18275-4_7
12. Calzavara, S., Grishchenko, I., Maffei, M.: HornDroid: Practical and sound static analysis of Android applications by SMT solving. In: EuroS&P '16. pp. 47–62. IEEE (2016). https://doi.org/10.1109/EuroSP.2016.16
13. Caniart, N., Fleury, E., Leroux, J., Zeitoun, M.: Accelerating interpolation-based model-checking. In: TACAS '08. pp. 428–442. LNCS 4963 (2008). https://doi.org/10.1007/978-3-540-78800-3_32

14. CHC Competition, https://chc-comp.github.io
15. Chen, Y., Kovács, L., Robillard, S.: Theory-specific reasoning about loops with arrays using Vampire. In: Vampire@IJCAR '16. pp. 16–32. EPiC 44 (2016). https://doi.org/10.29007/qk21
16. Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and Presburger arithmetic. In: CAV '98. pp. 268–279. LNCS 1427 (1998). https://doi.org/10.1007/BFb0028751
17. Dietsch, D., Heizmann, M., Hoenicke, J., Nutz, A., Podelski, A.: Ultimate TreeAutomizer (CHC-COMP tool description). In: HCVS/PERR@ETAPS '19. pp. 42–47. EPTCS 296 (2019). https://doi.org/10.4204/EPTCS.296.7
18. Dutertre, B.: Yices 2.2. In: CAV '14. pp. 737–744. LNCS 8559 (2014). https://doi.org/10.1007/978-3-319-08867-9_49
19. Ernst, G.: Loop verification with invariants and contracts. In: VMCAI '22. pp. 69–92. LNCS 13182 (2022). https://doi.org/10.1007/978-3-030-94583-1_4
20. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving constrained Horn clauses using syntax and data. In: FMCAD '18. pp. 1–9 (2018). https://doi.org/10.23919/FMCAD.2018.8603011
21. Finkel, A., Leroux, J.: How to compose Presburger-accelerations: Applications to broadcast protocols. In: FSTTCS '02. pp. 145–156. LNCS 2556 (2002). https://doi.org/10.1007/3-540-36206-1_14
22. Frohn, F., Giesl, J.: Proving non-termination via loop acceleration. In: FMCAD '19. pp. 221–230 (2019). https://doi.org/10.23919/FMCAD.2019.8894271
23. Frohn, F.: A calculus for modular loop acceleration. In: TACAS '20. pp. 58–76. LNCS 12078 (2020). https://doi.org/10.1007/978-3-030-45190-5_4
24. Frohn, F., Giesl, J.: Proving non-termination and lower runtime bounds with LoAT (system description). In: IJCAR '22. pp. 712–722. LNCS 13385 (2022). https://doi.org/10.1007/978-3-031-10769-6_41
25. Frohn, F., Giesl, J.: Proving non-termination by Acceleration Driven Clause Learning. In: CADE '23. LNCS (2023), to appear. Full version appeared in CoRR **abs/2304.10166**, https://doi.org/10.48550/arXiv.2304.10166.
26. Frohn, F., Giesl, J.: ADCL: Acceleration Driven Clause Learning for Constrained Horn Clauses. CoRR **abs/2303.01827** (2023). https://doi.org/10.48550/arXiv.2303.01827
27. Ganty, P., Iosif, R., Konečný, F.: Underapproximation of procedure summaries for integer programs. Int. J. Softw. Tools Technol. Transf. **19**(5), 565–584 (2017). https://doi.org/10.1007/s10009-016-0420-7
28. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting combinatorial search through randomization. In: AAAI '98. pp. 431–437 (1998), https://www.cs.cornell.edu/gomes/pdf/1998_gomes_aaai_iaai_boosting.pdf
29. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI '12. pp. 405–416 (2012). https://doi.org/10.1145/2254064.2254112
30. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: CAV '15. pp. 343–361. LNCS 9206 (2015). https://doi.org/10.1007/978-3-319-21690-4_20
31. Haase, C., Halfon, S.: Integer vector addition systems with states. In: RP '14. pp. 112–124. LNCS 8762 (2014). https://doi.org/10.1007/978-3-319-11439-2_9
32. Henzinger, T.A., Hottelier, T., Kovács, L., Rybalchenko, A.: Aligators for arrays (tool paper). In: LPAR '10. pp. 348–356. LNCS 6397 (2010). https://doi.org/10.1007/978-3-642-16242-8_25

33. Hoder, K., Bjørner, N.S.: Generalized property directed reachability. In: SAT '12. pp. 157–171. LNCS 7317 (2012). https://doi.org/10.1007/978-3-642-31612-8_13
34. Hojjat, H., Iosif, R., Konečný, F., Kuncak, V., Rümmer, P.: Accelerating interpolants. In: ATVA '12. pp. 187–202. LNCS 7561 (2012). https://doi.org/10.1007/978-3-642-33386-6_16
35. Hojjat, H., Rümmer, P.: The Eldarica Horn solver. In: FMCAD '18. pp. 1–7 (2018). https://doi.org/10.23919/FMCAD.2018.8603013
36. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: JayHorn: A framework for verifying Java programs. In: CAV '16. pp. 352–358. LNCS 9779 (2016). https://doi.org/10.1007/978-3-319-41528-4_19
37. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods Syst. Des. **48**(3), 175–205 (2016). https://doi.org/10.1007/s10703-016-0249-4
38. Kostyukov, Y., Mordvinov, D., Fedyukovich, G.: Beyond the elementary representations of program invariants over algebraic data types. In: PLDI '21. pp. 451–465 (2021). https://doi.org/10.1145/3453483.3454055
39. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: FASE '09. pp. 470–485. LNCS 5503 (2009). https://doi.org/10.1007/978-3-642-00593-0_33
40. Kroening, D., Lewis, M., Weissenbacher, G.: Under-approximating loops in C programs for fast counterexample detection. Formal Methods Syst. Des. **47**(1), 75–92 (2015). https://doi.org/10.1007/s10703-015-0228-1
41. libFAUDES Library, https://fgdes.tf.fau.de/faudes/index.html
42. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Information Processing Letters **47**(4), 173–180 (1993). https://doi.org/10.1016/0020-0190(93)90029-9
43. Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based verification for Rust programs. ACM Trans. Program. Lang. Syst. **43**(4), 15:1–15:54 (2021). https://doi.org/10.1145/3462205
44. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS '08. pp. 337–340. LNCS 4963 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
45. OEIS Foundation Inc.: Thue-Morse sequence. The On-Line Encyclopedia of Integer Sequences, published electronically at https://oeis.org/A010060
46. OEIS Foundation Inc.: First differences of Thue-Morse sequence. The On-Line Encyclopedia of Integer Sequences (1999), published electronically at https://oeis.org/A029883
47. Schrammel, P., Jeannet, B.: Logico-numerical abstract acceleration and application to the verification of data-flow programs. In: SAS '11. pp. 233–248. LNCS 6887 (2011). https://doi.org/10.1007/978-3-642-23702-7_19
48. Silverman, J., Kincaid, Z.: Loop summarization with rational vector addition systems. In: CAV '19. pp. 97–115. LNCS 11562 (2019). https://doi.org/10.1007/978-3-030-25543-5_7
49. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A cross-community infrastructure for logic solving. In: IJCAR '14. pp. 367–373. LNCS 8562 (2014). https://doi.org/10.1007/978-3-319-08587-6_28
50. Wesley, S., Christakis, M., Navas, J.A., Trefler, R.J., Wüstholz, V., Gurfinkel, A.: Verifying Solidity smart contracts via communication abstraction in SmartACE. In: VMCAI '22. pp. 425–449. LNCS 13182 (2022). https://doi.org/10.1007/978-3-030-94583-1_21
51. Zhu, H., Magill, S., Jagannathan, S.: A data-driven CHC solver. In: PLDI '18. pp. 707–721 (2018). https://doi.org/10.1145/3192366.3192416