Interpolation-Based Verification of Floating-Point Programs with Abstract CDCL *

Martin Brain[1], Vijay D'Silva[3], Alberto Griggio[2**],
Leopold Haller[1], and Daniel Kroening[1]

[1] University of Oxford
first.last@cs.ox.ac.uk
[2] Fondazione Bruno Kessler, Trento, Italy
griggio@fbk.eu
[3] University of California, Berkeley
vijayd@eecs.berkeley.edu

Abstract. One approach for SMT solvers to improve efficiency is to delegate reasoning to abstract domains. Solvers using abstract domains do not support interpolation and cannot be used for interpolation-based verification. We extend Abstract Conflict Driven Clause Learning (ACDCL) solvers with proof generation and interpolation. Our results lead to the first interpolation procedure for floating-point logic and subsequently, the first interpolation-based verifiers for programs with floating-point variables. We demonstrate the potential of this approach by verifying a number of programs which are challenging for current verification tools.

1 Introduction

Numeric software that manipulates floating-point variables is ubiquitous in automotive, avionic, medical, public transportation and other safety critical systems. The IEEE 754 standard defines the format of, operations on, and exceptions concerning floating-point computations. To alleviate the complexity of floating-point reasoning, some solvers use abstract domains to manipulate and approximate the semantics of formulae [2, 14, 22, 24].

In this paper, we study solvers that implement the *Abstract Conflict Driven Clause Learning* (ACDCL) algorithm [9]. ACDCL solvers lift the Conflict Driven Clause Learning (CDCL) algorithm in SAT solvers to operate on abstract domain elements instead of propositional formulae. To enable the use of ACDCL solvers in interpolation-based verification, we extend ACDCL with proof generation and interpolant construction. We apply our theoretical results to derive verifiers for programs with floating-point variables.

* Supported by the Toyota Motor Corporation, ERC project 280053, EPSRC project EP/J012564/1, and the FP7 STREP PINCETTE.
** Supported by Provincia Autonoma di Trento and the European Community's FP7/2007-2013 under grant agreement Marie Curie FP7 – PCOFUND-GA-2008-226070 "progetto Trentino", project ADAPTATION.

The intuition behind our work stems from the construction of propositional interpolants. Clause learning SAT solvers can generate resolution proofs [25] and interpolants can be constructed in time linear in the size of a proof [18]. We introduce ACDCL proofs, which extend propositional resolution with reasoning about abstract domain elements. Under certain conditions, discussed later, ACDCL proofs can be rewritten to obtain proofs with the structure generated by DPLL(T) solvers. Existing techniques can be used to construct interpolants from such proof [19]. The major difference between our work and existing work is not in the interpolation procedure we use but in the solver algorithm used to generate proofs. Extending ACDCL to generate proofs with the same structure as DPLL(T) solvers is useful because there are cases where DPLL(T) solvers time out out but an ACDCL solver does not [1].

Contributions and Contents In this paper, we present and evaluate the first interpolation-based verification procedures for programs with floating-point variables. Our work makes the following contributions.

1. Generation of proofs for ACDCL based on the notion of abstract resolution. Abstract resolution generalises resolution to accommodate formula manipulation in an abstract domain.
2. Sufficient conditions for computing interpolants from ACDCL proofs, and for linear-time interpolation in a theory.
3. The first interpolation-based verifiers for floating-point logic. We implement both the Bounded Model Checking-based interpolation algorithm of [18], and two variants of lazy abstraction with interpolants [20, 3].
4. Our implementations perform better than existing state-of-the-art verification tools on a set of small but challenging floating-point programs.

The paper is organised as follows: Section 2 contains a recap of ACDCL, and Section 3 presents our extension of resolution and of ACDCL to generate proofs. Our results on interpolation appear in Section 4, which includes a treatment of the issues arising in the floating point context. We present our empirical results in Section 5, followed by related work in Section 6.

2 Abstract Conflict Driven Clause Learning

We recall the abstract satisfaction framework, which allows us to study satisfiability problems in terms of lattices and transformers and is the basis for ACDCL. We refer the reader to [9] for a deeper treatment of ACDCL, and to [14] for an instantiation of ACDCL for floating-point reasoning.

2.1 The Abstract Satisfaction Framework

Logic We work with standard first-order notions of predicates, functions and terms. An *atomic predicate* is a predicate symbol composed with terms. A *literal* is an atomic predicate or its negation. A *clause* is a disjunction of literals, and a

cube is a conjunction of literals. A CNF formula is a conjunction of clauses, and one in DNF is a disjunction of cubes.

We assume a *satisfaction relation* $\models$ between structures in a set *Structs* and formulae. A structure σ is a *model* of φ if $\sigma \models \varphi$, otherwise, σ is a *countermodel*. A formula is *satisfiable* if it has a model and is *unsatisfiable* otherwise. The *satisfiability problem* is to determine whether a given formula is satisfiable. We write SAT for the satisfiability problem for propositional logic.

Lattices A *lattice* $(L, \sqsubseteq, \sqcup, \sqcap)$ is a partially ordered set with a meet and a join. The *powerset lattice* over a set X, written $(\wp(S), \subseteq, \cup, \cap)$, contains subsets of S order by inclusion. Two functions $f, g : Q \to L$ from a set Q to L can be *ordered pointwise*, denoted $f \sqsubseteq g$, if $f(x) \sqsubseteq g(x)$ holds for all x in Q. Functions on L also lift pointwise to $Q \to L$. The least and greatest fixed points of a monotone function f on a complete lattice will be denoted $\mathsf{lfp}(f)$ and $\mathsf{gfp}(f)$, respectively.

Let id_S be the identity function on a set S. A *Galois connection* between posets $(C, \sqsubseteq)$ and $(A, \preccurlyeq)$, written $(C, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \preccurlyeq)$, is a pair of monotone functions $\alpha : C \to A$ and $\gamma : A \to C$ satisfying the pointwise constraints $\alpha \circ \gamma \preccurlyeq id_A$ and $id_C \sqsubseteq \gamma \circ \alpha$.

Concrete Semantics of Formulae We recall a fixed point characterisation of satisfiability [9].The *concrete domain of structures* is $(\wp(\mathit{Structs}), \subseteq, \cup, \cap)$. A formula φ defines two *structure transformers*. The name structure transformers is used by analogy to state transformers and predicate transformers. Let X be a set of structures. The *model transformer* mods_φ removes all countermodels of φ from X, and the *conflict transformer* confs_φ adds all countermodels of φ to X.

$$\mathit{mods}_\varphi(X) \mathrel{\hat{=}} \{\sigma \in X \mid \sigma \models \varphi\} \quad \mathit{confs}_\varphi(X) \mathrel{\hat{=}} \{\sigma \in \mathit{Structs} \mid \sigma \not\models \varphi \text{ or } \sigma \in X\}$$

Properties of a formula can be expressed with transformers. The set of models of φ is $\mathit{mods}_\varphi(\mathit{Structs})$ and the set of countermodels of φ is $\mathit{confs}_\varphi(\emptyset)$.

Theorem 1. *The following statements are equivalent.*

1. *A formula φ is unsatisfiable.*
2. *The greatest fixed point $\mathsf{gfp}(\mathit{mods}_\varphi)$ contains no structures.*
3. *The least fixed point $\mathsf{lfp}(\mathit{confs}_\varphi)$ contains all structures.*

Applying the transformers above amounts to solving the ALL-SAT problem and is at least as hard as satisfiability. For efficiency, we use abstraction.

Abstract Satisfaction We overapproximate models and underapproximate countermodels. Let $(O, \sqsubseteq, \sqcup, \sqcap)$ be an overapproximation of the domain of structures and $(U, \preccurlyeq, \curlyvee, \curlywedge)$ be an underapproximation. The approximation is formalised by the Galois connections below. The orders $\sqsubseteq$ and $\preccurlyeq$ both refine set inclusion on structures. That is, $a \sqsubseteq b$ implies $\gamma(a) \subseteq \gamma(b)$, and $x \preccurlyeq y$ implies $\gamma(x) \subseteq \gamma(y)$.

$$(\wp(\mathit{Structs}), \subseteq) \underset{\alpha_O}{\overset{\gamma_O}{\leftrightarrows}} (O, \sqsubseteq) \qquad (\wp(\mathit{Structs}), \supseteq) \underset{\alpha_U}{\overset{\gamma_U}{\leftrightarrows}} (U, \succcurlyeq)$$

An *abstract model transformer* $amods_\varphi : O \to O$, and an *abstract conflict transformer* $aconfs_\varphi : U \to U$ satisfy the pointwise constraints below.

$$mods_\varphi \circ \gamma_O \subseteq \gamma_O \circ amods_\varphi \qquad\qquad confs_\varphi \circ \gamma_U \supseteq \gamma_U \circ aconfs_\varphi$$

The basic soundness result of abstract interpretation can be used to derive sound but incomplete satisfiability solvers.

Theorem 2. *A formula φ is unsatisfiable over a set of structures Structs if at least one of the conditions below hold.*

1. *The set $\gamma_O(\mathsf{gfp}(amods_\varphi))$ is empty.*
2. *The set $\gamma_U(\mathsf{lfp}(aconfs_\varphi))$ contains all structures.*

If $\mathsf{gfp}(amods_\varphi)$ concretises to the empty set, φ must be unsatisfiable. Due to imprecision in the abstraction, $\gamma_O(\mathsf{gfp}(amods_\varphi))$ may not be empty even if φ is unsatisfiable. Similar intuition applies to reasoning with $aconfs_\varphi$.

2.2 A Recap of ACDCL

Recent work has given an abstract interpretation characterisation of the clause learning algorithm in SAT solvers [9]. This characterisation builds upon the observation that the data structures and operations in propositional SAT solvers are defined entirely by the notion of a literal. Propositional literals are the generators of CNF formulae, partial assignments (the data structure for deduction) and clauses (used in learning). The unit rule, decisions, and conflict analysis, can all be formulated in terms of literals. If we can generalise the notion of a literal, all else follows. The work in [9] shows that *complementable meet irreducibles* are a mathematical generalisation of literals to abstract domains. We review this characterisation next.

Irreducible Elements Irreducible elements in a lattice cannot be derived from other elements using meets and joins. A lattice element x is *completely meet irreducible* if for all $X \subseteq L$, the equality $x = \bigsqcap X$ implies x is in X. The set of meet irreducibles of L is denoted $Irr_\sqcap(L)$. A *meet decomposition* is a function $mdc : L \to \wp(Irr_\sqcap(L))$ satisfying $x = \bigsqcap mdc(x)$ for all x. We shorten 'completely meet irreducible' to 'meet irreducible' in this paper. A meet irreducible m of an abstract domain is *complementable* if there is an element $\overline{m}$ satisfying that $\neg\gamma(m) = \gamma(\overline{m})$. A domain *has complementable meet irreducibles* if every element is the meet of meet irreducibles, and every meet irreducible is complementable.

Domains to Logic ACDCL uses both abstract domain elements and formulae. We use lower case letters such as p, q, r for logical literals, φ, ψ for formulae, and m, n for domain elements. Our work applies to abstract domains that satisfy the requirements below. The first two conditions enforce that the semantics of logical formulae and abstract domain elements are both given in terms of a set of structures. The fourth condition ensure that meet irreducibles of the abstract domain can be represented by logical formulae. The formula representation is required to generate proofs. In Section 4.3 we show that choosing a representation is non-trivial.

Assumption 1 *Let $\mathcal{L}$ be set of formulae in the logic we consider and O be an abstract domain. We make the following assumptions.*

1. *Formulae in $\mathcal{L}$ are interpreted over structures in a set Structs.*
2. *The concretisation function γ is in $O \to \wp(Structs)$.*
3. *The abstract domain O has complementable meet irreducibles.*
4. *There exists a function $\langle\cdot\rangle : Irr_{\sqcap}(O) \to \mathcal{L}$ which maps every meet irreducible m to a formula $\langle m\rangle$ such that $\gamma(m)$ is the set of models of $\langle m\rangle$.*

We write P, Q, M, N for objects that are formulae or logical representations of meet irreducibles, and $\overline{P}$ denotes $\neg P$ if P is a formula, or $\langle\overline{m}\rangle$, if P is the representation of the meet irreducible m. Assumption 1 allows us to represent logical negations of abstract domain elements as clauses. We adopt the standard convention of writing clauses as sets of literals. Even if an abstract domain is not complemented, we can exploit meet irreducibles to represent the negation of a lattice element as a clause. The *clausal negation* of an abstract element a is the set $neg(a) \hat{=} \{\langle\overline{m}\rangle \mid m \in mdc(a)\}$.

Learning as Transformer Refinement ACDCL discovers regions of the search space that do not contain models of a formula and uses learning to navigate subsequent search away from such regions. An abstract element a is a *conflict* if $mods_{\varphi}(\gamma(a))$ is the empty set. The *best learning transformer* for a conflict a, defined below, prunes abstract elements using conflicts.

$$Learn_a : O \to O \qquad\qquad Learn_a \hat{=} x \mapsto \alpha(\gamma(x) \cap \neg\gamma(a))$$

A *learning transformer* is one that overapproximates the best learning transformer. A learning transformer removes countermodels from an abstract element, but may not remove all. ACDCL discovers conflicts, synthesises learning transformers, and uses these transformers to refine the analysis. We now elaborate on the details of conflict discovery and learning transformer synthesis.

The propositional unit rule asserts that if a region of the search space contains no model for all but one literal in a clause, every model of the clause must be a model of the remaining literal. The *abstract unit rule* lifts this intuition to abstract domains. Let θ be the clausal negation $neg(c)$ of some conflict c.

$$Unit_{\theta}(a) \hat{=} \begin{cases} \bot & \text{if } \gamma(p \sqcap a) = \emptyset \text{ for all } p \text{ in } \theta \\ a \sqcap q & \text{otherwise, if there exists } q \text{ in } \theta \\ & \quad \text{and for all } p \neq q \text{ in } \theta, \gamma(a \sqcap p) = \emptyset \\ a & \text{otherwise} \end{cases} \tag{1}$$

Learning corresponds to synthesising abstract unit rules from conflicts. ACDCL can be understood as generating a sequence of transformers, as below.

$$amods^0_{\varphi} \hat{=} \prod_{\theta \in \varphi} Unit_{\theta} \quad amods^{i+1}_{\varphi} \hat{=} amods^{i}_{\varphi} \sqcap Unit_{neg(c)}, \text{ for some conflict } c$$

ACDCL begins with unit rules for clauses in the formula, and alternates between two phases, *model search*, where conflicts are discovered, and *conflict analysis*,

where the conflicts are generalised. Eventually, a satisfying assignment is found, or the formula is shown to be unsatisfiable, or the precision limit of the abstract domain is reached with an inconclusive result, or, in certain cases, the procedure may not terminate. Algorithmic details of ACDCL relevant for proof generation are discussed in the next section.

3 Proofs from ACDCL

The contribution of this section is to generalise resolution to encode abstract domain reasoning, and extend ACDCL with proof generation.

3.1 Abstract Resolution

The resolution rule asserts that if the conjunction of $\theta \vee p$ with $\neg p \vee \psi$ is satisfiable, the clause $\theta \vee \psi$ must also be satisfiable. The rule is formulated entirely in terms of literals, and can be lifted to complementable meet irreducibles. We generalise resolution in two directions illustrated below.

Example 1. We consider a variable x interpreted as an interval, and write a constraint $x \in [0, \infty]$ as $x \geq 0$ for convenience. Of the three inferences below, standard resolution permits the first one.

$$\frac{\theta \vee \langle x \leq 0\rangle \quad \neg\langle x \leq 0\rangle \vee \psi}{\theta \vee \psi} \qquad \frac{\theta \vee \langle x \leq 0\rangle \quad \langle \overline{x \leq 0}\rangle \vee \psi}{\theta \vee \psi} \qquad \frac{\theta \vee \langle x \leq 0\rangle \quad \langle x \geq 1\rangle \vee \psi}{\theta \vee \psi}$$

The second inference uses complementable meet irreducibles for Boolean reasoning about certain abstract domain elements. The third inference requires theory reasoning, namely that $(x \leq 0) \sqcap (x \geq 1)$ is $\bot$.

We formalise the inferences above with an extension of the resolution rule that eliminates pairs of meet irreducibles. We encode theory reasoning by a *semantic resolution* rule that applies if a pair of elements reduce to bottom in an abstract domain

Definition 1 (Abstract Resolution). *Let θ and ψ be clauses. Abstract resolution consists of three rules. The literal resolution rule lRES is standard resolution, and mRES extends the standard rule to complementable meet irreducibles.*

$$\frac{\theta \vee p \quad \neg p \vee \psi}{\theta \vee \psi}\ l\text{RES} \qquad \frac{\theta \vee \langle m\rangle \quad \langle \overline{m}\rangle \vee \psi}{\theta \vee \psi}\ m\text{RES}$$

The semantic resolution rule sRES below uses the meet in the abstract domain to eliminate elements.

$$\frac{\theta \vee \langle m\rangle \quad \langle n\rangle \vee \psi}{\theta \vee \psi}\ s\text{RES},\ \textit{if } m \sqcap n = \bot,$$

After applying resolution, a literal may occur multiple times in a resolvent if it occurs in both antecedents. When dealing with a theory, a resolvent may be of the form $\langle m \rangle \vee \langle n \rangle$, where m and n are meet irreducibles satisfying $m \sqsubseteq n$. Such a clause can be *semantically folded* to the equivalent clause $\langle n \rangle$. More generally, the semantic folding of a clause

Definition 2 (Folding). *The semantic folding of a clause θ is the clause*

$$s\text{FOLD}(\theta) \;\hat{=}\; \{\langle m \rangle \in \varphi \mid \;\nexists \langle n \rangle \in \varphi \text{ such that } m \sqsubseteq n\}$$

containing syntactic representations of the maximal elements of θ.

In addition to the abstract resolution rules, ACDCL solvers reason using conflicts. A conflict is a region of the space with no models, so its negation, when viewed as a formula is a tautology. The standard proof-theoretic treatment of conflicts in the SMT literature is to treat them as theory lemmas. We adopt the same convention.

Definition 3. *A theory lemma is a clause $\theta \vee \psi$ satisfying that θ is the clausal negation $neg(c)$ of an element satisfying $c \sqsubseteq aconfs_{\neg\psi}(\bot)$.*

Intuitively, c contains only countermodels of $\neg\psi$, so c "implies" ψ, and the contrapositive of this statement is a tautology that we encode as a clause.

Definition 4 (ACDCL proof). *Consider a* CNF *formula φ. The hypothesis rule* HYP *and lemma rule* LEMMA *are given below.*

$$\frac{}{\theta}\ \text{HYP},\ \textit{if } \theta \in \varphi \qquad \frac{}{\theta \vee P}\ \text{LEMMA},\ \textit{if} \left\{ \begin{array}{l} \theta \vee P \textit{ is a theory lemma,} \\ \textit{and } P \textit{ is a literal of } \varphi \\ \textit{or } P \textit{ is a meet irreducible} \end{array} \right\}$$

A clause θ is derived from a CNF *formula φ by* ACDCL *if θ is introduced by* HYP *or* LEMMA*, or if θ is derived by applying either the abstract resolution rules or semantic folding to clauses derived from φ by* ACDCL*. An* ACDCL *refutation is an* ACDCL *derivation of $\bot$.*

Theorem 3 extends the soundness of resolution to ACDCL proofs.

Theorem 3. *If there exists an* ACDCL *derivation of a clause θ from a formula φ then $\varphi \models \theta$.*

Proof. The proof is by induction on the structure of an ACDCL derivation. For the base case, consider the hypothesis and lemma rules.

1. Clauses introduced by HYP belong to φ.
2. A clause $\theta \vee P$ with $\theta = neg(c)$ is derived from an element $c \sqsubseteq aconfs_{\neg P}(\bot)$, so by the soundness of abstract transformers, $\gamma(c) \subseteq confs_{\neg P}(\emptyset)$, and by negation $\neg\gamma(c) \supseteq \neg confs_{\neg P}(\emptyset)$, and by negation of the formula, $\neg\gamma(c) \cup mods_P(Structs) = Structs$, so $neg(c) \vee P$ is valid.

For the induction step, we assume that the theorem holds for clauses derived by ACDCL. The case for lRES is standard, and the reasoning for mRES and sRES are similar, so we only consider mRES. Let σ be a model of $c_1 \vee \langle l_1 \rangle$ and of $c_2 \vee \langle l_2 \rangle$. There are three cases. If σ does not satisfy $\langle l_1 \rangle$ or $\langle l_2 \rangle$, it satisfies $c_1 \vee c_2$. If σ does not satisfy l_1, it must satisfy c_1, hence satisfies $c_1 \vee c_2$. The case for not satisfying l_2 is identical. Note σ cannot satisfy both l_1 and l_2 because they are logical representations of a complemented pair. □

Corollary 1 (Soundness). *If there exists an* ACDCL *refutation for φ, then φ is unsatisfiable.*

3.2 Proofs from runs of ACDCL

We now discuss the algorithmic details of ACDCL and show how the algorithm can be extended with proof generation. The algorithm operates on a sequence of meet-irreducibles called an *abstract trail*.

Model search Model search can be viewed as a way to guide conflict analysis. The meet of elements in the trail, say a, represents the region considered for model search. If the set $\gamma(a)$ contains a model, the fixed point $\mathsf{gfp}(\mathit{amods}^i_\varphi \sqcap \lambda x.a)$ will be non-empty. If this fixed point is strictly smaller than a, new meet irreducibles are added to the trail. Elements added to the trail are deduced facts and are associated with a *reason*. The reason is either a subformula of φ, or a formula representing the learned transformer $\mathit{Unit}_{\mathit{neg}(c)}$.

Example 2. Consider the following CNF formula φ in linear integer arithmetic:

$$\varphi \hat{=} (x \geq 3) \wedge (x + y \leq 5) \wedge ((x \leq 0) \vee (y \geq 6))$$

ACDCL over the interval abstract domain produces the following trail during model search:

i	trail_i	$\mathit{reason}[i]$
1 :	$\langle x \geq 3 \rangle$	$\leftarrow (x \geq 3)$
2 :	$\langle y \leq 2 \rangle$	$\leftarrow (x + y \leq 5)$
3 :	$\bot$	$\leftarrow (x \leq 0) \vee (y \geq 6)$

The meet irreducible $\langle y \leq 2 \rangle$ is deduced in step 2 from the trail $\langle x \geq 3 \rangle$ and the reason $(x + y \leq 5)$. A conflict is discovered in step 3.

If a conflict is not found, ACDCL makes decisions. A decision is a meet irreducible that when conjoined with the current trail, yields a strictly smaller element. If a does represent the empty set, ACDCL enters the conflict analysis phase.

Conflict Analysis The goal of conflict analysis is to generalise a conflict a to a larger, still conflicting region. Proof generation only takes place during conflict analysis phase, so we discuss it in greater detail. Conflict analysis is detailed by the uncoloured lines in Algorithm 1 (see [14] for details). Given a reason r for a conflict a the analysis uses an transformer $\mathit{aconfs}_{r,a}$ satisfying two properties:

1. $\mathit{aconfs}_{r,a}$ is sound in the sense that it underapproximates the transformer $\lambda x.\ \mathit{confs}_r(x) \cup \gamma(a)$, and
2. $\mathit{aconfs}_{r,a}$ generalises, meaning that $\mathit{aconfs}_{r,a}(b) \sqsupseteq a$ for all elements b.

Conflict analysis steps backwards through the trail and generalises each meet irreducible by applying the conflict transformer with respect to the associated reason. The generalised result is stored in the *marking* array. An invariant of the algorithm is that after each main loop iteration, the meet of elements in *marking* is a conflict. This conflict is used to synthesise a learning transformer. If $\top$ is not conflicting after the analysis, a backjump undoes part of the trail to return to an earlier, non-conflicting state from which model search continues.

Proof generation Proof construction mirrors the construction of resolution proofs from runs of a propositional SAT solver [25]. We walk backwards along the trail and identify proofs steps encoding the reasoning that was performed. The main difference to the propositional case is that an ACDCL proof has to account for the reasoning performed by *aconfs* in the abstract domain.

The proof-producing extension of ACDCL conflict analysis is given by the coloured lines of Algorithm 1. The algorithm uses an array called *proof* to map clauses to proof fragments. We reuse the names of proof rules as functions that construct proof steps. In the case of resolution rules, the second argument is the resolved literal, and the other arguments are antecedents. We write RES for lRES and mRES, because both encode Boolean reasoning, so the distinction is not important for correctness of the algorithm.

The *proof* array is initialized by associating each clause in the input formula with an HYP application. Lines 11-17 constructs a proof to justify that $\mathit{marking}[i]$ can be deduced from q by applying the abstract unit rule to $\mathit{reason}[i]$. Line 18 constructs a proof for the propagation of $\mathit{marking}[i]$ in the trail. The piecewise proofs in the *pl* array are consolidated in lines 23-24 to derive a proof for the learnt clause.

Example 3. We revisit the formula φ and trail in Example 2 and illustrate both conflict analysis and proof construction. In this example, we do distinguish between lRES and mRES. Abstract conflict analysis starts from index 3 in the trail. Suppose that applying $\mathit{aconfs}_{\mathit{reason}[3]}$ to $\bot$ yields the set of meet irreducibles $q \hat{=} \{\langle y \leq 5\rangle, \langle x \geq 1\rangle\}$. Then *marking* is updated as below.

$$\mathit{marking}[1] \leftarrow \langle x \geq 1\rangle \qquad\qquad \mathit{marking}[2] \leftarrow \langle y \leq 5\rangle$$

The element $\mathit{reason}[3]$ is unit under q, with $\mathit{amods}_{(y\geq 6)}(q) = \bot$. We obtain the proof below

$$r \;\hat{=}\; \dfrac{\dfrac{}{(x \leq 0) \vee (y \geq 6)}\ \text{HYP} \qquad \dfrac{}{\overline{\langle x \geq 1\rangle} \vee \overline{\langle y \leq 5\rangle} \vee \neg(y \geq 6)}\ \text{LEMMA}}{(x < 0) \vee \overline{\langle x \geq 1\rangle} \vee \overline{\langle y \leq 5\rangle}}\ l\text{RES}$$

which we extend to $\mathcal{P}_3$:

$$\mathcal{P}_3 \;\hat{=}\; \dfrac{\dfrac{}{\neg(x \leq 0) \vee \overline{\langle x \geq 1\rangle} \vee \overline{\langle y \leq 5\rangle}}\ \text{LEMMA} \qquad r}{\overline{\langle x \geq 1\rangle} \vee \overline{\langle y \leq 5\rangle}}\ l\text{RES}$$

At the next iteration, we have $\mathit{marking}[2] = \langle y \leq 5 \rangle$. Applying $\mathit{aconfs}_{\mathit{reason}[2]}$ to $\mathit{marking}[2]$ returns $q \hat{=} \{\langle x \geq 0 \rangle\}$. Then $\mathit{marking}[1]$ is set to $\langle x \geq 1 \rangle \sqcap \langle x \geq 0 \rangle = \langle x \geq 1 \rangle$, and the following proof $\mathcal{P}_2$ is generated:

$$\mathcal{P}_2 \;\hat{=}\; \dfrac{\dfrac{}{\neg(x+y \leq 5) \vee \langle \overline{x \geq 0} \rangle \vee \langle y \leq 5 \rangle}\ \textsc{lemma} \qquad \dfrac{}{(x+y \leq 5)}\ \textsc{hyp}}{\langle \overline{x \geq 0} \rangle \vee \langle y \leq 5 \rangle}\ l\textsc{res}$$

Finally, at the last iteration, we have $\mathit{marking}[1] = \langle x \geq 1 \rangle$, and applying $\mathit{aconfs}_{\mathit{reason}[1]}$ to $\mathit{marking}[1]$ returns $\top$. The following proof $\mathcal{P}_1$ is generated:

$$\mathcal{P}_1 \;\hat{=}\; \dfrac{\dfrac{}{\neg(x \geq 3) \vee \langle x \geq 1 \rangle}\ \textsc{lemma} \qquad \dfrac{}{(x \geq 3)}\ \textsc{hyp}}{\langle x \geq 1 \rangle}\ l\textsc{res}$$

The final refutation $\mathcal{P}_a$ is obtained by combining $\mathcal{P}_3, \mathcal{P}_2$ and $\mathcal{P}_1$ as follows:

$$\mathcal{P}_a \;\hat{=}\; \dfrac{\dfrac{\mathcal{P}_3 \quad \mathcal{P}_2}{\langle \overline{x \geq 1} \rangle}\ s\textsc{res} \qquad \mathcal{P}_1}{\bot}\ s\textsc{res},$$

where in the first sRES application we applied sFOLD to eliminate $\langle \overline{x \geq 0} \rangle$.

We conclude the section with a correctness proof.

Theorem 4. *Let learnt be the clause returned by the proof-producing abstract conflict analysis algorithm of* ACDCL *(Algorithm 1). Then proof*$[\mathit{learnt}]$ *is an abstract resolution proof for learnt.*

Proof. Assume by induction that $\mathit{proof}[\mathit{reason}[i]]$ is an abstract resolution proof for $\mathit{reason}[i]$, for each non-decision position i in the trail.

First, we show that the LEMMA and RES applications at lines 14, 17 and 18 are correct. For the LEMMAs, the side conditions hold by the correctness of *amods* and *aconfs* and by the definition of the abstract unit rule (1). For the RES at line 17, $\overline{l} \in p$ by construction, and $l \in p_i$ by the inductive hypothesis. Similarly, for the RES at line 18, $u \in p_i$ because $u \in \mathit{proof}[\mathit{reason}[i]]$ by the inductive hypothesis and $u \notin \mathit{unitreason}$ by construction. As a consequence of the correctness of such LEMMA and RES applications, the proof p_i generated at line 18 is a correct abstract resolution proof for the clause $\mathit{marking}[i] \vee \overline{q}$ (since all literals $L \in \mathit{reason}[i] \setminus q$ are eliminated by the sequence of resolutions at line 17). Moreover, $q \subseteq \{c \mid \exists 1 \leq j < i$ such that $\mathit{marking}[j] \sqsubseteq c\}$. Because of this, in the applications of sRES at line 24, $l \in p$ and P contains a literal l_2 such that $\overline{l_2} \sqsubseteq \overline{l}$. Therefore, the side conditions of sRES are satisfied. In order to prove the theorem, it remains to show that the literals of P not involved in the sequence of sRES applications of line 24 are exactly those in the set $\{\overline{\mathit{marking}[i]} \mid 1 \leq i \leq |\mathit{trail}|$ and $\mathit{marking}[i] \neq \top\}$. Since the elements of *marking* are meet irreducibles, after the update $\mathit{marking}[r] \leftarrow \mathit{marking}[r] \sqcap c$ at line 10,

Algorithm 1: ACDCL proof generation during abstract conflict analysis.

```
1  abstract-conflict-analysis(trail, reason, proof)
2      i ← |trail|; marking ← {1 ↦ ⊤, ..., (i − 1) ↦ ⊤, i ↦ ⊥};
3      pl ← nil;
4      loop
5          if marking[i] ≠ ⊤ then
6              a ← ⨅_{1≤j<i} trail[j];
7              q ← aconfs_{reason[i],a}(marking[i]);
8              foreach c in mdc(q) do
9                  r ← smallest index r' s.t. trail_{r'} ⊑ c;
10                 marking[r] ← marking[r] ⊓ c ;
11             unitreason ← nil; u ← ⊤;
12             foreach l in reason[i] do
13                 if amods_l(q) = ∅ then
14                     if l̄ ∉ q then unitreason ← unitreason : (l, LEMMA(l̄ ∨ q̄));
15                 else u ← l;
16             p_i ← proof[reason[i]];
17             foreach (l, p) in unitreason do p_i ← RES(p_i, l, p);
18             if u ≠ ⊤ then p_i ← RES(p_i, u, LEMMA(ū ∨ q̄ ∨ marking[i]));
19             pl ← pl : (marking[i], p_i);
20         marking[i] ← ⊤; i ← i − 1;
21         if stopping-criterion(trail, marking) then
22             confl ← ⨅_{1≤i≤|trail|} marking[i]; learnt ← ¬confl;
23             (_, P) ← pl[1];
24             foreach (l, p) in pl[2 ... |pl|] do P ← sRES(P, l, p);
25             proof[learnt] ← P;
26             return learnt;
```

either $marking[r]$ is set to $c \in q$, or $marking[r]$ was already set to an element c' of the result q' of *amods* of a previous iteration of the loop of Algorithm 1. In both cases, the new value of $marking[r]$ will occur in some proof in the list pl, and hence in the root of P. Also the old value of $marking[r]$ before the update at line 10 will occur in some proof in the list pl, if it was not $\top$. However, such values will not occur in the root of P thanks to the use of sFOLD in the applications of sRES. □

Corollary 2. *Let φ be a* CNF *formula. If* ACDCL *can prove the unsatisfiability of φ, then there exists an abstract resolution refutation for it.*

4 Interpolation for ACDCL

The contribution of this section is sufficient conditions for deriving interpolants from ACDCL proofs. We show how to reuse interpolant constructions for resolution proofs as well as proofs from DPLL(T) solvers to compute interpolants. This allows us to take advantage of the large body of results about interpolation in

SAT and SMT, while still retaining the performance benefits that ACDCL might have over DPLL(T) (see e.g. [1]).

4.1 ACDCL and DPLL(T) proofs

DPLL(T) solvers generate Boolean resolution proof with leaves that are input clauses or theory lemmas [19]. We define such proofs in our setting below.

Definition 5. *Given a* CNF *formula φ and a clause θ, a* DPLL(T) *proof of θ from φ is an abstract resolution proof containing no* sRES *applications.*

It should not come as a surprise that an abstract resolution proof can be transformed into a DPLL(T) proof satisfying the definition above. The transformation can be achieved by replacing sRES steps by a combination of mRES and LEMMA steps, as indicated below.

1. An sRES step involving $\langle l_1 \rangle$ and $\langle \overline{l_1} \rangle$ can be replaced by an mRES step.
2. An sRES step involving $\langle l_1 \rangle$ and $\langle l_2 \rangle$ can be replaced by a combination of two mRES and one LEMMA steps as below.

$$\dfrac{c_1 \vee \langle l_1 \rangle \quad c_2 \vee \langle l_2 \rangle}{c_1 \vee c_2}\, s\text{RES} \longrightarrow \dfrac{c_1 \vee \langle l_1 \rangle \qquad \dfrac{c_2 \vee \langle l_2 \rangle \quad \dfrac{}{\langle \overline{l_1} \rangle \vee \langle \overline{l_2} \rangle}\,\text{LEMMA}}{c_2 \vee \langle \overline{l_1} \rangle}\, m\text{RES}}{c_1 \vee c_2}\, m\text{RES}$$

3. An sFOLD step which removes an element $\langle l_2 \rangle$ because of an element $\langle l_1 \rangle$ satisfying $l_2 \sqsubseteq l_1$ can be rewritten as follows:

$$\dfrac{c \vee \langle l_2 \rangle \vee \langle l_1 \rangle \quad \dfrac{}{\langle \overline{l_2} \rangle \vee \langle l_1 \rangle}\,\text{LEMMA}}{c \vee \langle l_1 \rangle}\, m\text{RES}$$

Example 4. Consider again the formula φ and the refutation of Example 3. We convert it into a DPLL(T) proof with the transformation below.

$$\dfrac{\dfrac{\dfrac{\mathcal{P}_3 \quad \mathcal{P}_2}{\langle \overline{x \geq 1} \rangle}\, s\text{RES} \qquad \mathcal{P}_1}{\bot}\, s\text{RES}}{} \longrightarrow \dfrac{\dfrac{\mathcal{P}_3 \quad \mathcal{P}_2}{\langle \overline{x \geq 0} \rangle \vee \langle \overline{x \geq 1} \rangle}\, m\text{RES} \quad \dfrac{}{\langle x \geq 0 \rangle \vee \langle \overline{x \geq 1} \rangle}\,\text{LEMMA}}{\langle \overline{x \geq 1} \rangle}\, m\text{RES} \qquad \mathcal{P}_1}{\bot}\, m\text{RES}$$

4.2 Generation of interpolants

Constructing a DPLL(T) refutation from an abstract resolution refutation is the first step towards using existing interpolation algorithms like e.g. [19] with ACDCL. Such algorithms do not typically apply to arbitrary DPLL(T) proofs but require proofs to satisfy a syntactic condition commonly called *colourability*.[4].

Definition 6 (Colourability). *Let Σ be a set of symbols, let t be a term in a theory T, and let* $\mathsf{syms}(t)$ *be the set of symbols which occur in t and are uninterpreted in T. Then t is* Σ-colourable *iff* $\mathsf{syms}(t) \subseteq \Sigma$. *Given two formulas A and B in T, t is* A-colourable *if it is* $\mathsf{syms}(A)$*-colourable, and* B-colourable *if it is* $\mathsf{syms}(B)$*-colourable. If t is* $\mathsf{syms}(A) \cup \mathsf{syms}(B)$*-colourable but neither A-colourable nor B-colourable, t is* AB-mixed.

Instantiating ACDCL to work on abstract domains that do not allow AB-mixed terms enables interpolant generation for theories in which interpolation exists for conjunctions of literals. A more interesting case is to wonder whether it is possible to use ACDCL to compute interpolants for theories for which there is no known efficient interpolation procedure. The lemma below provides sufficient conditions on proof structure.

Lemma 1. *Let $\mathcal{P}_{\text{DPLL(T)}}$ be a* DPLL(T) *proof generated from an abstract resolution refutation for a formula $\varphi_A \wedge \varphi_B$ in a given theory T. If all the lemmas occurring in $\mathcal{P}_{\text{DPLL(T)}}$ are either A-colourable or B-colourable, then it is possible to compute an interpolant I for (φ_A, φ_B) from $\mathcal{P}_{\text{DPLL(T)}}$.*

Proof. Let

$$\psi_A \mathrel{\hat{=}} \varphi_A \wedge \bigwedge \{c \text{ is an } A\text{-colorable lemma of } \mathcal{P}_{\text{DPLL(T)}}\}$$
$$\psi_B \mathrel{\hat{=}} \varphi_B \wedge \bigwedge \{c \text{ is a } B\text{-colorable lemma of } \mathcal{P}_{\text{DPLL(T)}}\}$$

By the hypothesis, each lemma in $\mathcal{P}_{\text{DPLL(T)}}$ occurs in either ψ_A or ψ_B. Therefore, $\psi_A \wedge \psi_B$ is propositionally unsatisfiable, and $\mathcal{P}_{\text{DPLL(T)}}$ is a Boolean resolution refutation for $\psi_A \wedge \psi_B$. Thus, we can compute an interpolant I for (ψ_A, ψ_B) by applying an off-the-shelf Boolean interpolation algorithm to $\mathcal{P}_{\text{DPLL(T)}}$. Since the lemmas of $\mathcal{P}_{\text{DPLL(T)}}$ are by definition valid clauses in the theory T, ψ_A and ψ_B are logically equivalent to φ_A and φ_B in T. Therefore, I is an interpolant also for (φ_A, φ_B). □

One candidate for satisfying the conditions of Lemma 1 is to use a Cartesian abstract domain because every meet irreducible represents a predicate with one variable and can be coloured. Domain structure alone is insufficient because the conflict transformer must also respect the colorability requirement. We say a conflict transformer *aconfs* is *locality preserving* with respect to a formula $\varphi_A \wedge \varphi_B$ if for all colorable θ and elements a, all elements in the meet decomposition of $aconfs_\theta(a)$ are A-colorable or all are B-colorable.

[4] Preprocessing to enforce colourability in restricted cases is known [5]

Corollary 3. *If* ACDCL *is instantiated over a Cartesian domain and it produces an abstract resolution refutation $\mathcal{P}_a$ for an unsatisfiable formula $\varphi_A \wedge \varphi_B$, then an interpolant I for (φ_A, φ_B) can be computed from $\mathcal{P}_a$.*

Proof. In a Cartesian domain, complementable elements contain only one variable, and so they are always colorable. Therefore, $\mathcal{P}_a$ does not contain AB-mixed terms. Let $\mathcal{P}_{\text{DPLL(T)}}$ be a DPLL(T) refutation corresponding to $\mathcal{P}_a$. By the side conditions of LEMMA rule, lemmas in $\mathcal{P}_{\text{DPLL(T)}}$ consist of some complementable elements and at most one literal occurring in either φ_A or φ_B (or both). Therefore, assuming the conflict transformer is locality preserving, all the lemmas in $\mathcal{P}_{\text{DPLL(T)}}$ are colorable. By Lemma 1, then, we can compute an interpolant for (φ_A, φ_B) from $\mathcal{P}_{\text{DPLL(T)}}$. □

Example 5. We give an example showing that not all abstract resolution proofs are amenable to interpolation with existing DPLL(T)-based algorithms. Consider the following pair of formulas in linear arithmetic:

$$\varphi_A \hat{=} (x_3 + y_1 \leq x_1 + x_2) \wedge (x_1 \leq x_3) \wedge (x_2 \leq 0)$$
$$\varphi_B \hat{=} (z_1 \leq y_1) \wedge (1 \leq z_1)$$

An interpolant for (φ_A, φ_B) is the formula $(y_1 \leq 0)$.

Suppose that ACDCL is instantiated over the non-Cartesian abstract domain of octagons.A run of ACDCL might produce the following trail:

i	$trail_i$	$reason[i]$
1 :	$\langle -x_1 + x_3 \geq 0 \rangle$	$\leftarrow (x_1 \leq x_3)$
2 :	$\langle y_1 - z_1 \geq 0 \rangle$	$\leftarrow (z_1 \leq y_1)$
3 :	$\langle x_2 - z_1 \geq 0 \rangle$	$\leftarrow (x_3 + y_1 \leq x_1 + x_2)$
4 :	$\langle x_2 \geq 1 \rangle$	$\leftarrow (1 \leq z_1)$
5 :	$\bot$	$\leftarrow (x_2 \leq 0)$

A DPLL(T) proof for this trail is the following:

$$P \hat{=} \dfrac{\dfrac{\dfrac{\dfrac{P_5 \quad P_4}{\langle x_2 - z_1 \geq 0 \rangle} \quad P_3}{\langle -x_1 + x_3 \geq 0 \rangle \vee \langle y_1 - z_1 \geq 0 \rangle} \quad P_2}{\langle -x_1 + x_3 \geq 0 \rangle} \quad P_1}{\bot}$$

where:

$$P_5 \hat{=} \frac{\langle x_2 \geq 1 \rangle \vee \neg(x_2 \leq 0) \quad (x_2 \leq 0)}{\langle x_2 \geq 1 \rangle} \qquad P_4 \hat{=} \frac{\langle x_2 \geq 1 \rangle \vee \langle x_2 - z_1 \geq 0 \rangle \vee \neg(1 \leq z_1) \quad (1 \leq z_1)}{\langle x_2 \geq 1 \rangle \vee \langle x_2 - z_1 \geq 0 \rangle}$$

$$P_3 \hat{=} \frac{\langle -x_1 + x_3 \geq 0 \rangle \vee \langle y_1 - z_1 \geq 0 \rangle \vee \neg(x_3 + y_1 \leq x_1 + x_2) \vee \langle x_2 - z_1 \geq 0 \rangle \quad (x_3 + y_1 \leq x_1 + x_2)}{\langle -x_1 + x_3 \geq 0 \rangle \vee \langle y_1 - z_1 \geq 0 \rangle \vee \langle x_2 - z_1 \geq 0 \rangle}$$

$$P_2 \hat{=} \frac{\langle y_1 - z_1 \geq 0 \rangle \vee \neg(z_1 \leq y_1) \quad (z_1 \leq y_1)}{\langle y_1 - z_1 \geq 0 \rangle} \qquad P_1 \hat{=} \frac{\langle -x_1 + x_3 \geq 0 \rangle \vee \neg(x_1 \leq x_3) \quad (x_1 \leq x_3)}{\langle -x_1 + x_3 \geq 0 \rangle}$$

Since some of the leaves of P contain both A-colorable and B-colorable atoms, Boolean interpolation algorithms are not applicable to it. Moreover, P contains also the AB-mixed atom $\langle x_2 - z_1 \geq 0 \rangle$, which prevents also the use of off-the-shelf DPLL(T)-based interpolation algorithms for linear arithmetic (e.g. [19]).

4.3 An interpolation procedure for Floating Point Arithmetic

Using Corollary 3, we build a complete interpolation procedure for floating-point arithmetic (FPA), by instantiating ACDCL over the interval abstract domain for floating-point variables [14].

Floating Point Arithmetic Floating-point numbers are approximate representations of the reals that allow for fixed size bit-vector encoding. A floating-point number represents a real number as a triple of positive integers (s, m, e), consisting of a *sign bit* s taken from the set of Booleans $\{0, 1\}$, a *significand* m and an *exponent* e. Its real interpretation is given by $(-1)^s \cdot m \cdot 2^e$. A floating-point format determines the number of bits used for encoding significand and exponent. For a given format, we define $\mathbb{F}$ to be the set of all floating-point numbers plus the *special values* positive infinity $+\infty$, negative infinity $-\infty$, and NaN, which represents an invalid arithmetic result. *Terms* in FPA are constructed from floating-point variables, constants, standard arithmetic operators and special operators such as square roots and combined multiply-accumulate operations. Most operations are parameterized by one of five rounding modes. The result of floating-point operations is defined to be the real result (computed with 'infinite precision') rounded to a floating-point number using the chosen rounding mode. *Formulas* in FPA are Boolean combinations of predicates over floating-point terms. In addition to the standard equality predicate $=$, FPA offers a number of floating-point specific predicates including a special floating-point equality $=_{\mathbb{F}}$, and floating-point specific arithmetic inequalities $<$ and $\leq$. Since these operators approximate real comparisons they have unusual properties. For example, every comparison with the value NaN returns false, therefore $=_{\mathbb{F}}$ is not reflexive since $NaN =_{\mathbb{F}} NaN$ does not hold.

ACDCL-based interpolation for FPA We build our interpolation procedure upon FP-ACDCL, a sound and complete ACDCL-based satisfiability algorithm for FPA presented in [14]. More specifically, we instantiate ACDCL over the Cartesian abstract domain of intervals of floating-point values. In order to to this, we define a total order $\preceq$ over all floating-point values, including special values such as NaN. In particular, $\preceq$ is such that NaN is the minimum element, $-0 \preceq 0$, and $f_1 \preceq f_2 \iff f_1 \leq f_2$ in all other cases. Meet irreducibles in this domain are half-open intervals, which we denote with $\langle x \prec f \rangle$ or $\langle x \succeq f \rangle$ for a variable x and a floating-point value f.

We extend FP-ACDCL with proof-generation capabilities, and compute the interpolants using existing off-the-shelf proof-based interpolation algorithm for propositional logic (such as e.g. [19]), as described in the previous sections. The only thing to observe here is that, in general, the computed interpolants will contain predicates corresponding to some meet irreducibles $\langle x \prec f \rangle$, which are

not part of the signature of FPA as defined above. However, we can eliminate such predicates with a post-processing step on the generated interpolant, simply by replacing them with equivalent formulas in FPA. Notice that, because of the unusual properties of operations in FPA, in general a single meet irreducible cannot be represented by a single atom in FPA, but non-atomic formulas are needed. For example, the equivalent of $\langle x \preceq -0 \rangle$ in the syntax of FPA is the formula $(x = NaN) \vee ((x \leq 0) \wedge \neg(x = +0))$.

Example 6. Consider the following two formulas φ_A and φ_B in FPA (where "$\cdot_e$" denotes an operation with a "round to nearest even" rounding mode):

$$\varphi_A \hat{=} (x \geq 1.0) \wedge (x +_e y \leq 1.1)$$
$$\varphi_B \hat{=} (z \geq 0.2) \wedge (z < 0.22) \wedge (z *_e y > 0.05).$$

Suppose that FP-ACDCL generates the following DPLL(T) proof P for $\varphi_A \wedge \varphi_B$:

$$P \hat{=} \dfrac{\dfrac{\dfrac{\dfrac{P_5 \quad P_4}{\overline{\langle x \succeq 1.0\rangle} \vee \overline{\langle z \succeq 0.2\rangle} \vee \langle z \succeq 0.22\rangle} \; P_3}{\overline{\langle x \succeq 1.0\rangle} \vee \langle z \succeq 0.22\rangle} \; P_2}{\overline{\langle x \succeq 1.0\rangle}} \; P_1}{\bot}$$

where:

$$P_5 \hat{=} \dfrac{\neg(x +_e y \leq 1.1) \vee \overline{\langle x \succeq 1.0\rangle} \vee \overline{\langle y \succeq 0.1001\sim\rangle} \quad (x +_e y \leq 1.1)}{\overline{\langle x \succeq 1.0\rangle} \vee \overline{\langle y \succeq 0.1001\sim\rangle}}$$

$$P_4 \hat{=} \dfrac{\neg(z *_e y > 0.05) \vee \overline{\langle z \succeq 0.2\rangle} \vee \langle y \succeq 0.1001\sim\rangle \vee \langle z \succeq 0.22\rangle \quad (z *_e y > 0.05)}{\overline{\langle z \succeq 0.2\rangle} \vee \langle y \succeq 0.1001\sim\rangle \vee \langle z \succeq 0.22\rangle}$$

$$P_3 \hat{=} \dfrac{\neg(z \geq 0.2) \vee \langle z \succeq 0.2\rangle \quad (z \geq 0.2)}{\langle z \succeq 0.2\rangle}$$

$$P_2 \hat{=} \dfrac{\neg(z < 0.22) \vee \overline{\langle z \succeq 0.22\rangle} \quad (z < 0.22)}{\overline{\langle z \succeq 0.22\rangle}} \qquad P_1 \hat{=} \dfrac{\neg(x \geq 1.0) \vee \langle x \succeq 1.0\rangle \quad (x \geq 1.0)}{\langle x \succeq 1.0\rangle}$$

By applying the Boolean interpolation algorithm of [19] to $\mathcal{P}_{\text{DPLL(T)}}$, we obtain the interpolant $I \hat{=} \overline{\langle y \succeq 1.001\sim\rangle}$, which is equivalent to the FPA formula $\neg(y \geq 1.001\sim)$.

5 Evaluation

In order to evaluate the utility of our interpolation procedure for FPA, we have implemented several interpolation-based program verifiers, and performed experiments on a number of small but challenging floating-point programs. In this section, we present the results of our experimental evaluation.

5.1 Implementation

Interpolating Decision Procedure We have implemented our interpolating decision procedure within the MATHSAT5 SMT solver [4]. Details of the ACDCL solver for floating-point intervals are given in [14]. We have extended this solver with proof generation and interpolation. The implementation allows to choose among three different propositional interpolation algorithms for constructing interpolants from ACDCL proofs, and it also provides the option to combine ACDCL-based interpolation with the simple procedure based on inlining "definitional equalities" described in [12], which was shown to be particularly effective for formulas arising in software verification.

Program Verifiers We have implemented three different program verifiers based on interpolants. The first one, called "Monolithic" here, is the procedure proposed by McMillan in [18], which uses interpolants for computing overapproximations of postimages in symbolic transition systems for verifying circuits. The two others are variants of the "lazy abstraction with interpolants" algorithm of [20] for the verification of imperative sequential programs. We have implemented the original algorithm as described in [20] (called "Impact"), as well as the variant proposed in [3], which combines Impact with techniques inspired by the IC3 algorithm (called "TreeIC3+ITP" in [3], and simply "Impact with IC3-like strenghtening" here).[5]

5.2 Experimental Results and Discussion

Benchmarks We use three sets of benchmarks to demonstrate the range of application of floating-point interpolation. The first set of benchmarks, dcblock-simple, is derived from a simple filter in the CSound audio processing system. The programs contain infinite loops with a per-cycle input and non-linear arithmetic. Assertions check the variable ranges for each iteration. Proving correctness requires the verification system to be able to reason about the 'eventual' behaviour of the code.

The second set of benchmarks, rangevMain, is based on a widely-used iterative algorithm for computing square roots [21]. The main loop always terminates, but the number of steps is determined by the input and the initial guess, which are both non-deterministic. Proving properties of the result after the loop requires finding consequences of the loop invariant and reasoning about non-linear behaviour including division.

The final set of benchmarks, test, are synthetic tests, which require accurate reasoning about floating-point semantics. These demonstrate the limitation of using the 'standard model' of floating point [21] to convert the analysis into a non-linear real decision problem. Termination of loops and the reachability and

[5] Notice that in the TreeIC3+ITP algorithm of [3], interpolants are combined with underapproximated preimage computations based on quantifier elimination. Here, we only use interpolants, since we do not have an effective quantifier elimination procedure for FPA.

truth of assertions in these benchmarks require precise reasoning about floating-point arithmetic, including rounding and loss of precision.

Experimental Setup We present a comparison of interpolation-based verification using our technique with model checking and conventional abstract interpreters. We compare with SatAbs [6] (release 3.2 with Boom revision 201), a model checker that implements predicate abstraction, and Wolverine [17] (revision 69), an interpolant-based model checker. To the authors' knowledge, these are the only model checking tools that support bit-precise reasoning about floating point. In both cases, this is realized via 'bit-blasting', i.e., a translation to bit-vectors. We also compare with the commercial abstract interpretation systems Fluctuat [8] (version 3.1228) and Astrée [7] (version 12.10). In all cases, tools were run with their default options. We suspect that with expert assistance in their configuration, particularly the abstract interpretation tools, results could likely be improved.

The experiments were run on a 2.83 GHz Intel Core2 Q9550 using Fedora Core 17. Each experiment was limited to 1200 seconds and 3 GB RAM and was run sequentially to avoid inaccuracies due to cache and memory contention.

	TreeIC3+ITP	Model Checking		Abstract Interpretation	
		SatAbs	Wolverine	Fluctuat	Astrée
dcblock-simple-1	117.25	TO	MO	UN	**0.18**
dcblock-simple-2	117.47	TO	MO	UN	**0.19**
dcblock-simple-3	2.31	TO	MO	UN	**0.20**
dcblock-simple-4	727.26	TO	MO	UN	**0.15**
rangevMain1	**0.23**	TO	MO	UN	UN
rangevMain2	**0.23**	TO	MO	UN	UN
rangevMain2b	**0.17**	TO	MO	UN	UN
rangevMain5	**0.28**	TO	MO	UN	UN
rangevMain10	**0.34**	TO	MO	UN	UN
test1	**0.01**	2.16	TO	UN	UN
test2	0.90	**0.13**	MO	UN	UN
test3	6.89	**0.14**	9.94	UN	UN
test4	23.67	**0.13**	TO	UN	UN

Table 1. IMPACT with IC3-like strengthening vs. ACDCL and other tools.

Results and Discussion Table 1 compares Impact with IC3-like strengthening and equality inlining based on ACDCL with state-of-the-art research model checkers and commercial abstract interpretation tools. Times are recorded in seconds, where "TO" and "MO" denote experiments that reached the time and memory limits, respectively. Here, "UN" denotes experiments where safety could not be proven due to limitations of the abstraction. Further comparisons between different interpolation-based verification algorithms and between different interpolation schemes may be found in the appendix.

From the results it is clear that interpolation-based verification using ACDCL is a powerful technique that can verify a range of programs that are beyond the reach of current tools. Further discussion may be found in the appendix.

The performance of SatAbs and Wolverine shows the limitations of 'bit blasting' as an approach to deciding FPA theories. As they are bit precise, the test benchmarks can be handled, but they are unable to generate sufficiently concise invariants to allow other benchmarks to be verified. Conversely, the abstract interpretation tools are very fast (there were no runs that took more than 1 second) but in almost all cases they could not verify the assertions. The benchmarks that Astrée verified were likely due to having explicit domains for digital filters. Experimenting with the number of loop unrollings and the widening operators used may yield positive results as in some cases the computed ranges were close or were clearly converging before widening to the full interval.

6 Related Work

Our work resides in the context of interpolating SMT solvers. There is a performance gap between solvers and their interpolating counterparts, as well as a theoretical gap because proof generation is not well understood within all solver architectures. Interpolating solvers for first-order theories with use-cases in verification were introduced by McMillan [19], who follows the DPLL(T) paradigm, and supports linear rational arithmetic and integer difference arithmetic.

Interpolation frameworks have been developed for first-order theories by controlling the structure of proofs in a superposition-based theorem prover [15]. Another framework for computing interpolants in extensions of a base theory with additional symbols and axioms, by exploting interpolation algorithms for the base theory appeared in [23]. This paper presents a framework for ACDCL procedures, with an instantiation for the floating-point solver in [14].

The challenge addressed by our work is to study interpolation for a solver in which the notion of a proof is not obvious. The same challenge was addressed in [16] "lifting" propositional interpolants to equality logic in solvers that used Boolean encoding of equality formulae, and in [12] to derive bit-vector interpolants from interpolants for propositional logic and linear integer arithmetic. Our work differs from these by its focus on abstract interpretation-based solvers. We believe that our work is the first to attempt interpolation and proof generation in an abstract interpretation-based solver.

We now summarize applications of interpolating solvers in program analysis. The first application of an interpolating solver in software verification was for predicate discovery in Blast [19]. Wolverine [17] and the analyser in Sec. 5 implement the Impact algorithm [20]. We use abstract interpretation as building-block inside an interpolating solver. Conversely, interpolation is used in an abstract interpreter in [13] for automatically refining abstract interpretations. ACDCL can be applied directly to programs by extending the logic supported by the solver with fixed-point operators [10].

Finally, the combination of abstract interpretation and decidable logics for invariant generation has been recently explored by Garoche et al. [11]. In [22] constraint programming techniques are used for refining abstract interpretations of floating-point programs.

7 Conclusion

One approach to improving performance of decision procedures is to delegate some reasoning to an abstract domain. However, solvers that use abstract domains do not support interpolation and proof generation. We have presented proof generation and interpolation techniques for the family of ACDCL solvers, in which all reasoning is performed within an abstract domain. We have built upon these techniques to implement the first interpolation-based verifiers for programs with floating-point variables, and demonstrated that our verifiers extend the range of what can be automatically verified.

We observe a curious reversal of traditional roles in ours and related work. Abstract interpretation has historically been applied to reason about programs, while proofs and interpolation have been developed in a decision procedure context. We have however used abstract interpretation to design our decision procedure and interpolation to design our program verifier. The broad question for extending this line of work is to identify further techniques from abstract interpretation that can improve decision procedures, and to import techniques from decision procedures to develop program verifiers.

References

1. M. Brain, V. D'Silva, L. Haller, A. Griggio, and D. Kroening. An abstract interpretation of DPLL(T). In *Proc. of Verification, Model Checking and Abstract Interpretation*, volume 7737 of *LNCS*, pages 455–475. Springer, 2013.
2. A. Brillout, D. Kroening, and T. Wahl. Mixed abstractions for floating-point arithmetic. In *Proc. of Formal Methods in Computer-Aided Design*, pages 69–76. IEEE Computer Society Press, 2009.
3. A. Cimatti and A. Griggio. Software model checking via IC3. In *Proc. of Computer Aided Verification*, LNCS. Springer, 2012.
4. A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *LNCS*. Springer, 2013.
5. A. Cimatti, A. Griggio, and R. Sebastiani. Efficient generation of Craig interpolants in satisfiability modulo theories. *ACM Transactions on Computational Logic*, 12(1):7, 2010.
6. E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in Systems Design*, 25(2-3):105–127, 2004.
7. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *Proc. of the European Symposium on Programming*, volume 3444, pages 21–30. Springer, 2005.

8. D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *Proc. of the Workshop on Formal Methods for Industrial Critical Systems*, LNCS, pages 53–69. Springer, 2009.
9. V. D'Silva, L. Haller, and D. Kroening. Abstract conflict driven learning. In *Proc. of Principles of Programming Languages*, pages 143–154, 2013.
10. V. D'Silva, L. Haller, D. Kroening, and M. Tautschnig. Numeric bounds analysis with conflict-driven learning. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, pages 48–63, 2012.
11. P.-L. Garoche, T. Kahsai, and C. Tinelli. Invariant stream generators using automatic abstract transformers based on a decidable logic. *CoRR*, abs/1205.3758, 2012.
12. A. Griggio. Effective word-level interpolation for software verification. In *Proc. of Formal Methods in Computer-Aided Design*, 2011.
13. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 443–458. Springer, 2008.
14. L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding floating-point logic with systematic abstraction. In *Proc. of Formal Methods in Computer-Aided Design*, pages 131–140, 2012.
15. L. Kovacs and A. Voronkov. Interpolation and symbol elimination. In *Proc. of Automated Deduction*, volume 5663 of *LNCS*, pages 199–213. Springer, 2009.
16. D. Kroening and G. Weissenbacher. Lifting propositional interpolants to the word-level. In *FMCAD*, pages 85–89. IEEE, 2007.
17. D. Kroening and G. Weissenbacher. Interpolation-based software verification with Wolverine. In *CAV*, volume 6806, pages 573–578. Springer, 2011.
18. K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. of Computer Aided Verification*, volume 2725, pages 1–13. Springer, 2003.
19. K. L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.
20. K. L. McMillan. Lazy abstraction with interpolants. In *Proc. of Computer Aided Verification*, pages 123–136. Springer, 2006.
21. J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.
22. O. Ponsini, C. Michel, and M. Rueher. Refining abstract interpretation based value analysis with constraint programming techniques. In *Constraint Programming*, volume 7514, pages 593–607. Springer, 2012.
23. N. Totla and T. Wies. Complete instantiation-based interpolation. In *Proc. of Principles of Programming Languages*, pages 537–548. ACM Press, 2013.
24. C. Truchet, M. Pelleau, and F. Benhamou. Abstract domains for constraint programming, with the example of octagons. In *Symbolic and Numeric Algorithms for Scientific Computing*, pages 72–79, 2010.
25. L. Zhang and S. Malik. The quest for efficient Boolean satisfiability solvers. In *Proc. of Computer Aided Verification*, pages 17–36. Springer, 2002.