Form Methods Syst Des (2014) 45:213–245
DOI 10.1007/s10703-013-0203-7

Deciding floating-point logic with abstract conflict driven clause learning

Martin Brain · Vijay D'Silva · Alberto Griggio · Leopold Haller · Daniel Kroening

Published online: 20 December 2013
© The Author(s) 2013. This article is published with open access at Springerlink.com

Abstract We present a bit-precise decision procedure for the theory of floating-point arithmetic. The core of our approach is a non-trivial, lattice-theoretic generalisation of the conflict-driven clause learning algorithm in modern SAT solvers to lattice-based abstractions. We use floating-point intervals to reason about the ranges of variables, which allows us to directly handle arithmetic and is more efficient than encoding a formula as a bit-vector as in current floating-point solvers. Interval reasoning alone is incomplete, and we obtain completeness by developing a conflict analysis algorithm that reasons natively about intervals. We have implemented this method in the MATHSAT5 SMT solver and evaluated it on assertion checking problems that bound the values of program variables. Our new technique is faster than a bit-vector encoding approach on 80 % of the benchmarks, and is faster by one order of magnitude or more on 60 % of the benchmarks. The generalisation of CDCL we propose is widely applicable and can be used to derive abstraction-based SMT solvers for other theories.

Keywords Decision procedures · Floating-point logic · Abstract interpretation · SMT

Research supported by the Toyota Motor Corporation, ERC project 280053, EPSRC project EP/H017585/1, the FP7 STREP PINCETTE and DSTL under CDE Project 30713.
A. Griggio is 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.

M. Brain · D. Kroening
Computer Science Department, University of Oxford, Oxford, UK

V. D'Silva
University of California, Berkeley, USA

A. Griggio (✉)
Fondazione Bruno Kessler, Trento, Italy
e-mail: griggio@fbk.eu

L. Haller
Cadence Design Systems, Berkeley, USA

1 Introduction

Floating-point computations are pervasive in low-level control software and embedded applications. Such programs are frequently used in contexts where safety is critical, such as automotive and avionic applications. It is important to develop tools for accurate and scalable reasoning about programs manipulating floating-point variables.

Floating-point numbers have a dual nature that complicates complete logical reasoning. On the one hand, they are approximate representations of real numbers, which suggests reasoning about floating-point arithmetic using real arithmetic. On the other hand, floating-point numbers have a discrete, binary implementation, which suggests reasoning about them using bit-vector encodings and SAT solvers. Both approaches suffer from an explosion of cases that arises when considering the possible results of evaluating floating-point expressions.

An alternative to existing approaches is to use abstractions that enable efficient but imprecise reasoning. This approach is standard in static program analysis, including analyses that target safety critical embedded software with floating-point variables [6]. Our solver uses intervals for sound, efficient but imprecise reasoning about floating-point formulae. If imprecise reasoning cannot determine whether a formula is unsatisfiable, we use decisions to increase the precision of deduction, and then use conflict analysis to generalise the results of deduction. Our approach combines ideas from static program analysis with satisfiability algorithms and allows us to trade efficiency for precision in a demand-driven fashion. The approach is generic and can be used to implement solvers for other logics and theories. The rest of this section provides a more detailed overview of our technique and how it compares to existing techniques for designing decision procedures.

1.1 Discussion of floating-point solver architectures

We now discuss in detail a few different possibilities for designing an SMT solver for floating-point arithmetic. Interpreting a floating-point expression as a real arithmetic expression leads to incorrect conclusions because there are several cases where floating-point operations differ from real arithmetic operations. Encoding all these cases as constraints in a real arithmetic formula leads to large formulae containing several cases. Such formulae are difficult for real arithmetic solvers to handle.

A second approach to floating-point reasoning, called bit-blasting or propositional encoding, currently yields better performance than a real arithmetic encoding [11]. Floating-point operations are represented as circuits that are then translated into a Boolean formula that is solved by a propositional SAT solver. This approach enables precise modelling of floating-point semantics and allows high-performance SAT solvers to be reused to implement a floating-point solver. The disadvantage of this approach is that the SAT solver only has an operational view of arithmetic operations and must reason about individual bits in adder and multiplier circuits without the ability to simplify formulae using high-level, numeric reasoning.

A third approach is to use the popular DPLL(T) architecture [5]. The DPLL(T) architecture uses a SAT solver to reason about the Boolean structure of a formula and a specialised solver for conjunctions of theory literals. Thus, two efficient solvers for fragments of a theory are combined to obtain a solver for the theory. The first problem with using DPLL(T) is that we would still require a solver for conjunctions of literals in floating-point logic, and no off-the-shelf solution is available. A second issue is that, in some cases, separating Boolean

reasoning from theory reasoning is detrimental to performance [9, 51]. Details of the theory are not visible to the propositional solver and cannot guide search for a model, while information from previous runs are not available to the theory solver for conflict analysis.

The issues with DPLL(T) mentioned above are known and have fuelled research in *natural-domain* SMT procedures. The term 'natural-domain SMT' was first used by Cotton [19] but we use it for SMT procedures that perform all reasoning directly in a theory [19, 33, 47, 48, 51]. A fourth possibility is to develop a natural-domain floating-point solver which performs decisions, backtracking and learning using variables and atoms of the theory. The challenge in pursuing this approach is identifying which elements of the theory can be used for these operations, and developing efficient algorithms for propagation and learning.

In this paper, we pursue the fourth approach and develop a natural-domain SMT solver for reasoning about floating-point arithmetic formulae. We address the efficiency concerns highlighted above by developing a natural-domain SMT solver that supports imprecise reasoning. For insight into the operation of our solver, consider the formula

$$0.0 \leq x \wedge x \leq 10.0 \wedge y = x^5 \wedge y > 10^5$$

where the variables x and y have double-precision floating-point values. Interval propagation [54] tracks the range of each variable and can derive the fact $x \in [0.0, 10.0]$ from the first two constraints, which implies the fact $y \in [0.0, 100000.0]$ from the third constraint. This range is not compatible with the final conjunct $y > 10^5$, so the formula is unsatisfiable. The computation requires a fraction of a second with our interval solver. In contrast, translating the formula above into a bit-vector and invoking the SMT solver Z3 requires 16 minutes on a modern processor to prove that the formula is unsatisfiable.

The efficiency of interval reasoning comes at the cost of completeness. Consider the floating-point formula below.

$$z = y \wedge x = y \cdot z \wedge x < 0$$

After bit-vector encoding, the solver Z3 can decide satisfiability of this formula in a fraction of a second. The interval abstraction cannot represent relationships between the values of variables. Interval propagation will not deduce that y and z are either both positive or both negative. The interval solver cannot conclude that x must be positive and cannot show that the formula is unsatisfiable.

To recover completeness, we lift the Conflict Driven Clause Learning algorithm in SAT solvers to reason about intervals. Our solver uses intervals to make decisions, propagates intervals for deduction, and uses a conflict analysis over intervals to refine the results of interval propagation. Our algorithm is a strict, mathematical generalisation of propositional CDCL in that replacing floating-point intervals with partial assignments yields the original CDCL algorithm. Our approach is parametric, allowing for abstractions such as equality graphs, difference graphs, or linear inequalities to be used in place of floating-point intervals to obtain natural-domain CDCL algorithms for equality logic, difference logic, and linear arithmetic respectively.

Clause learning is not the only approach one may take to obtain a complete solver based on interval propagation. One may eliminate imprecision by splitting intervals into ranges that can be analysed without loss of precision. There are at least two ways to perform such splitting.

Splitting can be integrated in a DPLL(T) solver [4]. New propositions are required to represent intervals over ranges that do not occur explicitly in the original formula. Implementing good learning heuristics for this approach is difficult because the propositional learning algorithm is unaware of the intervals associated with these propositions.

Splitting can also be implemented in a natural-domain fashion. For the second example above, the solver can consider the cases $y < 0$ and $y \geq 0$, and can in each case conclude that x is positive. Such splitting yields a complete, natural-domain SMT solver that only manipulates intervals, but requires considering a potentially exponential number of cases. Moreover, the conclusions drawn from proving one case are not used to reason about another case, so the solver may repeatedly perform certain reasoning.

1.2 Content and contribution

In this paper, we present a Conflict Driven Clause Learning algorithm for floating-point logic. Our work exploits the insight presented in [28] that propositional SAT solvers internally operate on a lattice-based abstraction that overapproximates the space of possible solutions. We show how the FIRST-UIP learning algorithm [70] used in CDCL solvers can be lifted to a wider range of domains. This lifting is non-trivial since it has to address the additional complexity of abstractions for domains that go beyond propositional logic.

Contribution We make the following contributions.

1. We present a novel, natural-domain SMT solver for the theory of floating-point arithmetic. Our solver is based on a new perspective of SAT and SMT algorithms as techniques that manipulate lattice-based abstractions.
2. We lift the FIRST-UIP algorithm used for conflict analysis in modern SAT solvers to lattice-based abstractions. Our lifting enables lattice-based analyzers access to learning techniques that were hitherto limited to propositional SAT solvers.
3. We present a new implementation of our approach for floating-point logic as part of the MATHSAT5 framework. The implementation outperforms approaches based on bit-blasting significantly on our set of benchmarks.

Outline Section 2 provides a brief introduction to floating-point numbers and the theory of floating-point arithmetic. Section 3 recaps some of the formal background on lattices and abstract interpretation. Section 4 gives a high-level account of model search and conflict analysis over abstract domains. The main algorithmic contribution is presented in Sect. 5: A lifting of the FIRST-UIP algorithm to abstract domains. The implementation of our floating-point solver, the specific heuristics we used and experiments are discussed in Sect. 6. An extensive survey of related work from the areas of theorem proving, abstract interpretation, and decision procedures is given in Sect. 7.

2 A review of floating-point arithmetic

This section provides an informal introduction to floating-point numbers and some issues surrounding formal reasoning about floating-point. For a more in-depth treatment see [61].

2.1 Floating-point arithmetic

'Floating-point' is a style of encoding subsets of the rational numbers using bit-vectors of fixed width. The bit vectors are split into multiple, fixed size parts, including a fractional part (the *significand*) and a integer power by which it is multiplied (the *exponent*). Historically there were a number of different floating-point systems in common usage. This created significant problems when moving data between machines and made writing portable numerical software prohibitively difficult. A standardisation process lead to IEEE-754, which defines multiple floating point systems including their semantics and representation as bit strings. Since their introduction in 1985, the IEEE-754 formats have become the dominant floating-point system and the most common way of representing non-integer quantities. Most systems that do not comply with IEEE-754, such as some GPUs, simply implement a subset of its features. We focus on binary encodings of IEEE-754 for which the definitive reference is the standard [23].

The IEEE-754 binary encodings specify several different classes of numbers; normal, subnormal, zeros, infinities and "not a number" (*NaN*). Normal numbers are represented using a triple of unsigned binary numbers (s, e, m) in which s is the *sign bit* and always uses a single bit, e is the exponent and m is the significand and their width depends on the each format. The rational number represented by this pattern is given by the formulae:

$$(-1)^S \cdot 2^{e-bias} \cdot m$$

where *bias* is fixed by the format. An example of an IEEE-754 `binary16` floating-point normal number is given below.

$$\underbrace{\boxed{1}}_{s}\underbrace{\boxed{1|0|0|1|0}}_{e}\underbrace{\boxed{0|1|0|1|0|1|1|0|0|0}}_{m} = -1 \cdot 2^{18-15} \cdot (1 + 2^{-2} + 2^{-4} + 2^{-6} + 2^{-7}) = -10.6875$$

Note that the exponent is 5 bits, the significand is 10 bits and the *bias* is 15 $(2^{5-1} - 1)$. An exponent which is a sequence of 0s or a sequence of 1s represents one of the other types of number. Subnormal numbers have a 0 exponent and non-zero significand. They act as fixed-point numbers between the smallest normal number and zero, allowing the difference between two numbers to always be representable and improving the error bounds of computation. If both exponent and significand are 0, the number represents 0. Note that there are two floating-point numbers, $+0$ and -0 that both represent the rational value 0. These two numbers allow distinguishing between convergence to 0 from above and from below. Exponents that are all 1 with 0 significand represent infinity. Finally, an exponent of all 1 and a non-zero significand represent *NaN*.

IEEE-754 gives a precise and semi-operational semantics for common operations, including $+$, $-$, $*$, $/$ and $\sqrt{\cdot}$. For example, for normal and subnormal numbers and zeros, the standard specifies:

> ... every operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded.

Rounding depends on which rounding mode is used. IEEE-754 specifies five rounding modes, three directed (rounding up, rounding down, and rounding towards zero) and two round to nearest (with tie breaks to give an even significand or away from zero). Round to nearest with tie break to even is the default rounding mode.

We briefly discuss floating-point addition and multiplication. Floating-point addition is not associative even when restricted to normal numbers. Consider the two floating-point

expressions below where the numbers are represented by 32 bits with a 24 bit mantissa (the value 16777216 is 2^{24}).

$$(1 + 16777216) + -16777216 = 0 \qquad 1 + (16777216 + -16777216) = 1$$

25 bits are required to represent the sum of 1 and 2^{24}, so rounding is applied and 2^{24} is returned. Thus, the expression on the left evaluates to 0. No rounding is required to represent the result of evaluating subexpressions in the expression on the right, so the result is 1.

Floating-point multiplication is not associative either, as the example below demonstrates with 32-bit floating-point numbers.

$$(3 * 2049) * 8191 = 50350076 \qquad 3 * (2049 * 8191) = 50350080$$

When the result is a normal number there are tight bounds on the difference between the two orders of evaluation.

Moreover, floating-point addition does not distribute over multiplication because of rounding effects, as the example below demonstrates.

$$2049 * (8189 + 1) = 16781310 \qquad (2049 * 8189) + (2049 * 1) = 16781308$$

In the absence of associativity and distributivity, several standard algebraic approaches to reasoning about arithmetic expressions are inapplicable.

While few algebraic equivalences hold over floating-point numbers, ordering properties are generally preserved. All of the basic operations are piecewise monotonic (but not strictly monotonic) over normal and subnormal numbers. This means that techniques based on ordering, for example, interval abstractions, are particularly suitable for floating-point numbers.

2.2 Floating-point logic

The previous sub-section demonstrated that floating-point arithmetic does not behave like real arithmetic. It is important to develop specialised decision procedures for reasoning precisely about the results and properties of floating-point operations. The SMTLib theory of floating-point arithmetic (FPA) is a language for expressing constraints in terms of floating-point numbers, variables and operators. We refer to this theory as *floating-point logic* and review it in this section.

Terms A term in FPA is constructed from floating-point variables, constants, standard arithmetic operators and special operators. Examples of special operators include square roots and combined multiply-accumulate operations used in signal processing. These operations are parameterized by one of five rounding modes. The result of floating-point operations is defined to match IEEE-754; 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 $\leq$. These comparisons have to handle all classes of numbers. Normal and subnormal numbers are compared in the expected way. The two zeros, $+0$ and -0 are regarded as equal (despite having distinct floating-point representations) as they correspond to the same number. Infinities are respectively above ($+\infty$) and below ($-\infty$) all

of the preceding classes. Finally *NaN* is regarded to be unordered and incomparable to all floating point numbers, thus all comparisons involving *NaN*, including $NaN =_{\mathbb{F}} NaN$, are false. Thus standard equality, $=$, is reflexive but floating-point equality, $=_{\mathbb{F}}$ is not.

3 Background on lattices and abstraction

We now introduce abstract satisfaction, a lattice-theoretic framework for designing decision procedures. Abstract satisfaction is based on abstract interpretation, which provides a similar framework for reasoning about programs. An in-depth account of abstract satisfaction is given in [31].

3.1 Review of abstract interpretation

Abstract interpretation is formulated in terms *domains*, which are lattices equipped with monotone functions called *transformers*. The space of program behaviours is represented by a *concrete domain* and the behaviour of a program, called the *concrete semantics*, is characterised by a fixed-point expression. Checking properties of the concrete semantics is usually undecidable. An *abstract domain* is a lattice with transformers that can represent some but not all concrete behaviour. Checking properties of the abstract semantics is decidable but may be inaccurate.

Lattice and transformers A *poset* $(C, \sqsubseteq)$ is a set C equipped with a partial order. A *lattice* $(C, \sqsubseteq, \sqcup, \sqcap)$ is a partially ordered set with a greatest lower bound operator $\sqcup : C \times C \to C$, called *join*, and a least upper bound operator $\sqcap : C \times C \to C$, called *meet*. A lattice C is *complete* if every subset $X \subseteq C$ has a meet, denoted $\sqcap X$, and a join, denoted $\sqcup X$. The *powerset lattice* $(\wp(S), \subseteq)$ is the lattice of all subsets of S with the subset inclusion order.

A function $f : C \to A$ from a poset $(C, \sqsubseteq)$ to $(A, \preccurlyeq)$ is *monotone* if the order $x \sqsubseteq y$ implies the order $f(x) \preccurlyeq f(y)$ for all x and y in C. We use the term *transformer* for a monotone function $f : C \to C$ from a lattice to itself. A *fixed point* of a function $f : S \to S$ is an element satisfying the equality $f(x) = x$. On a poset, fixed points can be ordered. Tarski's fixed point theorem guarantees that transformers on complete lattices have least and greatest fixed points. We denote the *least fixed point* of f by $\mathsf{lfp}(f)$ and the *greatest fixed point* of f by $\mathsf{gfp}(f)$.

Approximation in abstract interpretation is formalised using pairs of functions. A *Galois connection* between posets, written $(C, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \preccurlyeq)$, is a pair of functions $\alpha : C \to A$ and $\gamma : A \to C$ satisfying the conditions below.

1. The functions α and γ are monotone.
2. For all x in C, $x \sqsubseteq \gamma(\alpha(x))$.
3. For all y in A, $\alpha(\gamma(y)) \preccurlyeq y$.

We can intuitively understand these conditions by interpreting $x \sqsubseteq y$ to mean that x has more information than y. For example, if x is a variable the bound $x \in [1, 3]$ is tighter than $x \in [0, 5]$ and provides more information about the range of x. The monotonicity condition above guarantees that order between elements is preserved. The second condition is about approximation and guarantees that every element in C can be represented by an element in A with some loss of information. The third condition is about precision and ensures that repeatedly moving from A to C does not increase information loss.

Abstract interpretation A *domain* $(C, \sqsubseteq, \sqcup, \sqcap, \{f_1, \ldots, f_k\})$ in abstract interpretation is a lattice equipped with transformers. The number of transformers depends on the details of the domain. A domain $(A, \preccurlyeq, \curlyvee, \curlywedge, \{af_1, \ldots, af_k\})$ is *a sound abstraction of* $(C, \sqsubseteq, \sqcup, \sqcap, \{f_1, \ldots, f_k\})$, if it contains a transformer af_i for each f_i and if the conditions below hold.

1. There is a Galois connection $(C, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \preccurlyeq)$ between the lattices.
2. Every pair of transformers f_i and af_i satisfies $f_i(\gamma(x)) \sqsubseteq \gamma(af_i(x))$ for all x in A.

The domain over C is called a *concrete domain* involving a *concrete lattice* and *concrete transformers*. The domain over A is called an *abstract domain* involving an *abstract lattice* and *abstract transformers*.

A *powerset domain* is one in which the lattice is of the form $\wp(S)$. Assume the concrete domain is a powerset domain. The abstract domain is *overapproximating* if $x \subseteq \gamma(\alpha(x))$ for all concrete elements x, and a transformer is *overapproximating* if $f_i(\gamma(y)) \subseteq \gamma(af_i(y))$ for all abstract elements y. The abstract domain is *underapproximating* if $x \supseteq \gamma(\alpha(x))$ for all concrete elements x, and a transformer is *underapproximating* if $\gamma(af_i(y)) \subseteq f_i(\gamma(y))$ for all abstract elements y.

Properties of programs are formalised using fixed points over transformers in a concrete domain. These properties can be approximated by computing the corresponding fixed point in the abstract domain. A key result of abstract interpretation is the *fixed point transfer* theorem showing that fixed points in the abstract domain approximate fixed points in the concrete domain.

3.2 Review of abstract satisfaction

We apply abstract interpretation to reason about formula satisfiability. The concrete domain we consider is a set of structures over which formulae are interpreted. We consider two different concrete semantics, one for deductive reasoning and one for abductive reasoning. Computing the concrete deductive semantics amounts to computing all models, while computing the abductive semantics amounts to computing all countermodels of a formula, with both computations being at least as hard as deciding satisfiability.

We review abstract satisfaction, an abstract interpretation framework for deciding satisfiability. The account that follows does not delve into details of specific logics. Let *Forms* be a set of *formulae* and *Structs* be a set of *structures*. We assume the interpretation of a formula over structures is specified in the standard manner by a relation $\models$, which is a subset of *Structs* $\times$ *Forms*. A structure σ is a *model of* φ if it satisfies $\sigma \models \varphi$ and is a *countermodel of* φ otherwise. A formula φ is *satisfiable* if it has a model.

We now develop a framework for characterising satisfiability via lattices and transformers. The concrete domain of structures, introduced below consists of all sets of structures. We introduce two *structure transformers*, which map between sets of structures and encode reasoning about models and countermodels of a formula.

Definition 1 The *concrete domain of structures*

$$\big(\wp(\mathit{Structs}), \subseteq, \cap, \cup, \{\mathit{mods}_\varphi, \mathit{confs}_\varphi \mid \varphi \in \mathit{Forms}\}\big)$$

is a powerset lattice of structures containing a *model transformer* mods_φ and a *conflict transformer* confs_φ defined below.

$$mods_\varphi \mathrel{\hat{=}} S \mapsto \{\sigma \in Structs \mid \sigma \text{ is in } S \text{ and } \sigma \models \varphi\}$$

$$confs_\varphi \mathrel{\hat{=}} S \mapsto \{\sigma \in Structs \mid \sigma \text{ is in } S \text{ or } \sigma \not\models \varphi\}$$

The model transformer maps a set of structures S to the largest subset of S that contains the same models as S. The conflict transformer (called the *universal countermodel transformer* in [28]) maps a set of structures S to the largest superset of S that contains the same models as S. The model transformer can be viewed as refining an overapproximation of a set of models while the conflict transformer can be viewed as generalising an underapproximation of a set of countermodels. Observe also that $mods_\varphi(Structs)$ is the set of all models of a formula and $confs_\varphi(\emptyset)$ is the set of all countermodels of a formula.

We introduce a concrete semantics below, which characterises satisfiability by fixed points over structure transformers. If a formula φ is unsatisfiable, it has no models, so $mods_\varphi(Structs)$ and the greatest fixed point of $mods_\varphi$ are the empty set. Moreover, every structure is a countermodel of φ, so $confs_\varphi(\emptyset)$ is equal to $Structs$ and so is the least fixed point of $confs_\varphi$.

Theorem 1 *The following statements are equivalent for a formula* φ.

1. *The formula* φ *is unsatisfiable.*
2. *The greatest fixed point* $\mathsf{gfp}(mods_\varphi)$ *is the empty set.*
3. *The least fixed point* $\mathsf{lfp}(confs_\varphi)$ *contains all structures.*

Our goal is to compute over- and underapproximations of the models and countermodels of a formula. If an overapproximation of the set of models of a formula is the empty set, the formula is unsatisfiable. If an underapproximation of the set of countermodels of a formula contains all structures, the formula is unsatisfiable. Abstract interpretation provides a framework for deriving these abstractions.

Consider an abstract domain

$$\left(A, \sqsubseteq, \sqcap, \sqcup, \{amods_\varphi, aconfs_\varphi | \varphi \in Forms\}\right)$$

that is a sound abstraction of the structures domain. If the domain is overapproximating, the inclusion $mods_\varphi(\gamma(x)) \subseteq \gamma(amods_\varphi(x))$ holds. If the domain is underapproximating, the inclusion $\gamma(aconfs_\varphi(x)) \subseteq confs_\varphi(\gamma(x))$ holds. The theorem below shows that we can iterate these transformers to obtain better approximations.

Theorem 2 *Let* $amods_\varphi$ *be an overapproximation of* $mods_\varphi$ *and* $aconfs_\varphi$ *be an underapproximation of* $confs_\varphi$.

1. *If* $\gamma(\mathsf{gfp}(amods_\varphi)) = \emptyset$ *then* φ *is unsatisfiable.*
2. *If* $\gamma(\mathsf{lfp}(aconfs_\varphi)) = Structs$ *then* φ *is unsatisfiable.*

Domains of floating-point numbers We introduce concrete and abstract domains of floating-point numbers. Let $\mathbb{F}$ be the set of all floating-point numbers and $(\wp(\mathbb{F}), \subseteq)$ be the set of subsets of floating-point numbers ordered by inclusion. Let *Vars* be the set of variables occurring in a formula. A *floating-point assignment* is a function $\sigma : Vars \to \mathbb{F}$. Floating-point assignments are the structures over which formulae in floating-point logic are interpreted. The *concrete domain of floating-point logic structures*

$$\left(\wp(Vars \to \mathbb{F}), \subseteq, \cap, \cup, \{mods_\varphi, confs_\varphi\}\right)$$

is defined over the lattice of sets of floating-point assignments with φ ranging over floating-point logic formulae.

We define the floating-point interval abstraction. Intervals approximate sets of numbers by their closest enclosing range. In addition to the arithmetic ordering $\leq$, the IEEE-754 standard dictates a total order $\preceq$ over all floating-point values, including special values such as *NaN*. The interval abstraction is defined with respect to this total order. The lattice $(\mathbb{I}, \sqsubseteq, \sqcap, \sqcup)$ of floating-point intervals is defined below. We write $\min_{\preceq}$ and $\max_{\preceq}$ for the minimum and maximum with respect to the $\preceq$ order.

1. The set of lattice elements is $\mathbb{I} \mathrel{\hat{=}} \{[a,b] | a, b \text{ are in } \mathbb{F} \text{ and } a \preceq b\} \cup \{\bot\}$.
2. The meet $\bot \sqcap y = y \sqcap \bot = \bot$ for all y. The meet $[a,b] \sqcap [c,d]$ is the interval $[\max_{\preceq}(a,c), \min_{\preceq}(b,d)]$ if $\max_{\preceq}(a,c) \preceq \min_{\preceq}(b,d)$ holds and is $\bot$ otherwise.
3. The join $\bot \sqcup y = y \sqcup \bot = y$. The join $[a,b] \sqcup [c,d]$ is $[\min_{\preceq}(a,c), \max_{\preceq}(b,d)]$.

We write $\top$ for the greatest element of $\mathbb{I}$. Given a set of variables *Vars*, the *interval domain* is the lattice

$$\big(Vars \to \mathbb{I}, \sqsubseteq, \sqcap, \sqcup, \{amods_\varphi, aconfs_\varphi\}\big)$$

with the components defined as below. We defer the definition of the transformers to the next two sections.

1. $f \sqsubseteq g$ for $f, g : Vars \to \mathbb{I}$ if $f(x) \sqsubseteq g(x)$ for all variables x.
2. $f \sqcap g$ is the function that maps a variable x to $f(x) \sqcap g(x)$.
3. $f \sqcup g$ is the function that maps a variable x to $f(x) \sqcup g(x)$.

We denote an element of the form $f : \{x, y\} \to \mathbb{I}$ as a tuple $\langle x{:}f(x), y{:}f(y)\rangle$ of variables paired with intervals. We omit from the tuple variables that map to $\top$. That is, if $f(x)$ is $\top$ and $f(y)$ is not, we write $\langle y{:}f(y)\rangle$. We follow the standard lattice-theoretic convention for overloading notation and write $\top$ for the greatest element of the interval domain. The interval lattice is related to the lattice of floating-point structures by a Galois connection.

$$\alpha : \wp(\mathbb{F}) \to \mathbb{I} \qquad \alpha(\emptyset) \mathrel{\hat{=}} \bot \qquad \alpha(S) \mathrel{\hat{=}} \big[\min_{\preceq}(S), \max_{\preceq}(S)\big], \quad \text{for } S \neq \emptyset$$

$$\gamma : \mathbb{I} \to \wp(\mathbb{F}) \qquad \gamma(\bot) \mathrel{\hat{=}} \emptyset \qquad \gamma(f) \mathrel{\hat{=}} \big\{\{x \mapsto v \mid x \in Vars, v \in f(x)\}\big\}$$

A standard fact in abstract interpretation is that the pair of functions defined above form a Galois connection [20].

4 Lifting CDCL to abstractions

In this section we show how the CDCL algorithm can be generalised to abstract domains. We call the result of this lifting Abstract CDCL (ACDCL). We focus on practical concerns. For a more formal perspective, and for soundness and completeness proofs see [30]. We first recall the propositional CDCL algorithm. Then we provide a detailed explanation of how each step can be lifted to abstract lattices. In our descriptions of CDCL and ACDCL we focus on a basic clause learning framework. Clause learning has been shown to be most salient aspect of the CDCL algorithm with regards to efficiency [65]. Propositional CDCL benefits from numerous further algorithmic and engineering advances [66], such as smart variable selection heuristics for decisions, effective data structures like watched literals, and algorithmic improvements such as restarts. Discussing all these improvements in a lattice based setting is beyond the scope of the paper. Some, for example restarts, lift to ACDCL in

a trivial manner while others, such as variable selection heuristics require domain-specific adaptations.

4.1 Review of propositional CDCL

The CDCL algorithm is shown in Algorithm 1. CDCL consists of two interacting phases, called model search and conflict analysis. *Model search*, shown in Algorithm 2, aims to find satisfying assignments for the formula. This process may fail and encounter a *conflicting* partial assignment, that is, a partial assignment that contains only countermodels. *Conflict analysis*, presented in Algorithm 4, extracts a general reason which is used to derive a new lemma over the search space in the form of a clause in a step called *learning*.

```
Input: set of clauses Φ
cdcl(Φ)
  π: partial assignment;
  tr: sequence of propositional assignments;
  reasons: partial function from Props ∪ {⊥} to Φ;
  π ← ∅; tr ← ϵ; reasons ← ∅;
  loop
    if modelSearch(π, tr, reasons, Φ) = SAT then
      return SAT;
    end
    c ← analyse(tr, reasons);
    Φ ← Φ ∪ {c};
    if not backjump(π, tr, c) then
      return UNSAT;
    end
```

Algorithm 1: The propositional CDCL algorithm

The fundamental datastructure used within the CDCL algorithm is the *partial assignment*, which is a partial function from a set of logical propositions *Props* to the Boolean truth constants $\mathbb{B} \mathrel{\hat{=}} \{\mathsf{t}, \mathsf{f}\}$. Partial assignments can be ordered by precision, and extended with a special symbol $\bot$, representing the empty set of models, to form the following lattice, as shown in Fig. 1.

$$(\mathit{PartAsg}, \sqsubseteq, \sqcap, \sqcup)$$

Partial assignments are an abstraction of the concrete lattice of propositional truth assignments $\wp(\mathit{Props} \to \mathbb{B})$. The abstraction and concretisation functions are given below.

$$\left(\wp(\mathit{Props} \to \mathbb{B}), \subseteq\right) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathit{PartAsg}, \sqsubseteq)$$

$$\alpha(\emptyset) \mathrel{\hat{=}} \bot \qquad \gamma(\bot) \mathrel{\hat{=}} \emptyset$$

$$\alpha(S) \mathrel{\hat{=}} \left\{p \mapsto v \mid \forall \sigma \in S.\, \sigma(p) = v\right\} \qquad \gamma(\pi) \mathrel{\hat{=}} \left\{\sigma \mid \forall p \mapsto v \in \pi.\, \sigma(p) = v\right\}$$

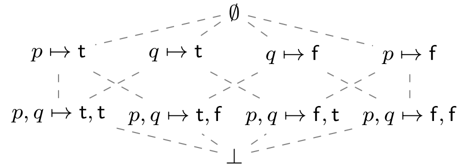


Fig. 1 The lattice of partial assignments *PartAsg*

Partial assignments are refined by applying the *unit rule* exhaustively in a step called Boolean Constraint Propagation (BCP). The unit rule is shown in Algorithm 3, and compares the literals of the clause with the current partial assignment. Given a clause and a partial assignment, the unit rule either returns a new variable assignment, a conflict element $\bot$, or the empty set, signifying that no new information could be deduced. We introduce some notation used in Algorithm 3: A literal l is in *positive phase* if it is of the form p for some proposition p; if it is of the form $\neg p$, it is in *negative phase*. The function *phase* returns the phase of a literal, i.e., $\mathit{phase}(l) = \mathsf{t}$ if l is in positive phase, and $\mathit{phase}(l) = \mathsf{f}$ otherwise. For a literal l, we denote its opposite phase literal by $\mathit{flip}(l)$, and by $\mathit{var}(l)$ the proposition p such that $l \in \{\neg p, p\}$.

From an abstract satisfaction perspective, a call to $\texttt{unit}\ (C, \cdot)$ computes an overapproximation of the model transformer mods_C. In fact, refining a partial assignment with the unit rule corresponds to the best abstract transformer of mods_C available in the partial assignments lattice [28]. We may alternately characterise the unit rule as a very natural abstract interpretation of the formula in which logical disjunction is interpreted as a join over the abstract lattice. For example for $C = p \vee \neg q \vee r$ and $\pi = \{p \mapsto \mathsf{f}, q \mapsto \mathsf{t}\}$:

$$\begin{aligned} \texttt{unit}\ (C, \pi) &= \texttt{unit}\ (p, \pi) \sqcup \texttt{unit}\ (\neg q, \pi) \sqcup \texttt{unit}\ (r, \pi) \\ &= \bot \sqcup \bot \sqcup \{r \mapsto \mathsf{t}\} = \{r \mapsto \mathsf{t}\} \end{aligned}$$

In addition to the partial assignment, the algorithm stores a *trail* tr, which is a sequence of singleton partial assignments of the form $\langle p{:}\mathsf{t}\rangle$ or $\langle p{:}\mathsf{f}\rangle$. We denote concatenation of an element to the trail by $\cdot$, and the ith element by $\mathit{tr}[i]$, e.g., the first element of the trail is $\mathit{tr}[1]$. The symbol ϵ denotes the empty trail. The trail stores the sequence of propositional assignments made during algorithm execution in chronological order. A *reasons* array maps a proposition p to a clause C if p was derived from C via the unit rule. If a conflict is derived, the reasons array maps the conflict element $\bot$ to the clause that was contradicted by the partial assignment.

The CDCL algorithm interleaves model search and conflict analysis as depicted in Fig. 2. Model search refines a partial assignment and extends the trail until either a satisfying assignment is found or a conflict is encountered. This is done in two ways: deduction with the unit rule identifies necessary consequences of the formula under the current partial assignments; *decisions* heuristically guess a value for unassigned propositions. If model search finds a satisfying assignment then the algorithm returns SAT.

If a conflict is encountered, conflict analysis uses the FIRST-UIP algorithm [70] which extracts a general conflict reason from the specific partial assignment that originally caused the conflict. The algorithm chooses as an initial generalisation R the elements of the conflicting partial assignment that contradicted the clause in $\mathit{reasons}[\bot]$. It then steps backwards through the trail, removes elements a of R, and replaces them with partial assignments that are sufficient to deduce a. At the end of every iteration of the generalisation loop, the contents of R are a sufficient reason for a conflict. This process continues until the first unique implication point (UIP) is reached (see [70] for details). From the final conflict reason R, a

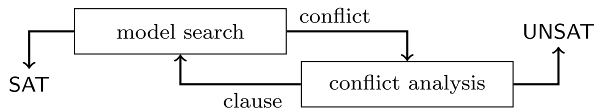


Fig. 2 A schematic depiction of the CDCL framework

clause is generated which expresses that R does not represent any models, and the result is added to the original formula φ. In future model search and conflict analysis steps this new *learnt clause* acts as a deduction shortcut.

After learning, backtracking resets the solver to an earlier state that is consistent with the newly learnt clause, or if this fails, the algorithm returns UNSAT. The stopping criterion during conflict analysis is coordinated with the backjumping step to ensure that the algorithm automatically explores a new region of the search space after backjumping (and thus avoids cycles). This mechanism is referred to as *backjumping with asserting clauses* [66, 67].

```
modelSearch(π, tr, reasons, Φ)
  loop
    repeat
      // Boolean Constraint Propagation
      foreach C ∈ Φ do
        q ← unit(C, π);
        if q = ⊥ then
          reasons[⊥] ← C;
          return CONFLICT;
        end
        if q = {p ↦ v} then
          π ← π ∪ {p ↦ v};
          tr ← tr · {p ↦ v};
          reasons[p] ← C;
        end
      end
    until π unchanged;
    if π is a complete assignment then return SAT;
    {p ↦ v} ← decide(π);
    π ← π ∪ {p ↦ v};
    tr ← tr · {p ↦ v};
    reasons[p] ← nil;
```

Algorithm 2: Propositional model search

4.2 Complementable meet irreducibles

As demonstrated above, partial assignments are an abstract lattice and the unit rule is an approximation of the *mods* transformer. We now identify specific properties of these objects that are necessary in order to lift the algorithm to other lattices. In propositional CDCL,

```
unit(C, π)
    u ← nil;
    foreach l ∈ C do
        p = var(l);
        if π is undefined at p then
            if u ≠ nil then return ∅;
            u ← l;
        end
        if phase(l) = π(p) then
            return ∅;
        end
    end
    if u = nil then return ⊥;
    if u = ¬p then
        return {p ↦ f};
    else
        return {p ↦ t};
    end
```

Algorithm 3: Propositional unit rule

```
analyse(tr, reasons)
    R ← {var(l) ↦ ¬phase(l) | l ∈ reasons[⊥]};
    for (i ← |tr|; UIPreached(R, tr); i ← i − 1) do
        a ← tr[i];
        if a ∉ R ∨ reasons[var(a)] = nil then continue;
        R ← R \ {a};
        R ← R ∪ {var(l) ↦ ¬phase(l) | l ∈ reasons[var(a)]};
    end
    return {p | p ↦ t ∈ R} ∪ {¬p | p ↦ f ∈ R};
```

Algorithm 4: FIRST-UIP conflict analysis

singleton partial assignments of the form $\{p \mapsto \mathsf{t}\}$ or $\{p \mapsto \mathsf{f}\}$ have a special role in the scope of the CDCL algorithm: (i) the unit rule returns singleton assignments as deduction results, (ii) they are the decision elements, i.e., a decision computes a meet between the current partial assignment and a singleton assignment, and (iii) they are stored in the trail datastructure *tr*.

In lattice theoretic terms, singleton assignments have a special property in that they cannot be expressed in terms of the meet over a set of other elements. A partial assignment $\{p \mapsto \mathsf{t}, q \mapsto \mathsf{f}\}$, for example, may be represented as the meet $\{p \mapsto \mathsf{t}\} \sqcap \{q \mapsto \mathsf{f}\}$, whereas the element $\{p \mapsto \mathsf{t}\}$ cannot be further decomposed in this way.

Definition 2 (Meet irreducibles) A *meet irreducible* in a complete lattice L is an element $m \in L$ different from $\top$ such that the following implication is valid.

$$\forall m_1, m_2 \in L \quad m_1 \sqcap m_2 = m \implies (m = m_1 \vee m = m_2).$$

Definition 3 (Meet Decomposition) A *meet decomposition* of an element $a \in A$ is a set $Q \subseteq A$ of meet irreducibles such that $\sqcap Q = a$.

In the case of partial assignments, meet irreducibles are exactly the singleton assignments. An important property of these elements is that they have precise complements. For example, a singleton assignment $\{p \mapsto \mathsf{t}\}$ represents the set of all propositional assignments where p is mapped to t. The complement of this set may be represented by the singleton assignment $\{p \mapsto \mathsf{f}\}$. This property is not shared by arbitrary partial assignments, e.g., the partial assignment $\{p \mapsto \mathsf{t}, q \mapsto \mathsf{t}\}$ represents a set of models whose complement has no precise representation in *PartAsg*.

Definition 4 (Complementable meet irreducibles) An abstract lattice A has *complementable meet irreducibles* if every meet irreducible $m \in A$ has a complementary meet irreducible $\overline{m} \in A$ such that $\gamma(m)$ is the set complement of $\gamma(\overline{m})$.

Example 1 The interval domain has complementable meet irreducibles. Consider the interval element $\langle x{:}[0.0, 5.3], y{:}[-3.6, 10.2]\rangle$. We may decompose the above element into meet irreducibles as follows.

$$\langle x \succeq 0.0\rangle \sqcap \langle x \preceq 5.3\rangle \sqcap \langle y \succeq -3.6\rangle \sqcap \langle y \preceq 10.2\rangle$$

Each of the elements of the decomposition above has a precise complement, e.g., $\overline{\langle x \succeq 0.0\rangle} = \langle x \prec 0.0\rangle$.

Meet irreducibles are also returned by the unit rule. A CNF formula φ can then be viewed as providing a set of sound approximations $\{\mathtt{unit}\,(C, \cdot) \mid C \in \varphi\}$ of the concrete model transformer $mods_\varphi$. We provide a general concept which lifts the relevant properties of the unit rule to lattices.

Definition 5 (Meet irreducible deduction) A *meet-irreducible deduction transformer* for a formula φ over an abstract domain A is a sound approximation $amods_\varphi : A \to A$ of $mods_\varphi$ such that for any $a \in A$, the element $amods_\varphi(a)$ is $\top$, $\bot$ or a meet irreducible.

Approximations of the model transformers are typically available in abstract domains in the form of strongest post-condition operators for logical guards. The required decomposition into meet irreducible deduction transformers can be achieved in practice by first applying a monolithic abstract model transformer and then computing a meet decomposition.

Example 2 Let *ded* be the best abstract transformer of $mods_\varphi$ over the intervals for the formula $\varphi = (-x = y)$, and let $\sigma = \langle x{:}[5.0, 10.0]\rangle$. We have that $ded(\sigma) = \sigma \sqcap \langle y : [-10.0, -5.0]\rangle$. We can decompose *ded* into a set of complementable rules $Ded = \{ded_x^l, ded_x^u, ded_y^l, ded_y^u\}$ s.t. $\sqcap Ded = ded$, and each of the elements of *Ded* infers a lower or an upper bound on x or y: $ded_x^l(\sigma) = \langle x \succeq 5.0\rangle$, $ded_x^u(\sigma) = \langle x \preceq 10.0\rangle$, $ded_y^l(\sigma) = \langle y \succeq -10.0\rangle$ and $ded_y^u(\sigma) = \langle y \preceq -5.0\rangle$.

Table 1 Propositional concepts and their abstract-satisfaction counterparts

Propositional CDCL	Abstract interpretation
Partial assignments	Abstract lattice with complementable meet-irreducibles
Singleton assignments	Meet irreducibles
Unit rule	Meet irreducible deduction transformer
CNF	Set of meet irreducible deduction transformers
Learnt clause	Learnt transformer

4.3 Abstract CDCL

We now show how CDCL may be lifted to abstract domains that have (i) complementable meet irreducibles (ii) an approximation of the concrete model transformer $mods_\varphi$ expressed in terms of a set of meet irreducible deduction transformers. We reinterpret the propositional algorithm as a lattice-based procedure using the correspondences listed in Table 1 to translate between the world of propositions and partial assignments and that of lattices and abstraction.

The resulting ACDCL framework is shown in Algorithm 5. Partial assignments are replaced by an element of the abstract domain, and the set of input clauses is replaced by a set of meet-irreducible deduction transformers. The *reasons* array maps elements of the trail to the transformers that were used to derive them.

```
Input: Set F of meet irreducible deduction transformers for φ over domain A.
acdcl(F)
    a: element of A;
    tr: sequence of meet irreducibles;
    reasons: partial function from trail indices and ⊥ to F;
    a ← ⊤; tr ← ε; reasons ← ∅;
    loop
        if aModelSearch(a, tr, reasons, F) = SAT then
            return SAT;
        end
        f ← aAnalyse(tr, reasons);
        F ← F ∪ {f};
        if not backjump(a, tr, f) then
            return UNSAT;
        end
```

Algorithm 5: The ACDCL algorithm

Abstract model search is shown in Algorithm 6. In place of BCP, a greatest fixed point over the transformers in F is computed. Narrowing [20] may be used to enforce convergence of the fixed point computation. A simple way to implement narrowing is simply to end the

loop early after a fixed number of iterations. Whenever a meet irreducible is inferred, it is put on the trail and the transformer that was used to infer it is stored as its reason.

```
aModelSearch(a, tr, reasons, F)
  loop
    repeat
      // Greatest fixed point
      foreach f ∈ F do
        q ← f(a);
        if q = ⊥ then
          reasons[⊥] ← f;
          return CONFLICT;
        end
        if q ≠ ⊤ then
          a ← a ⊓ q;
          tr ← tr · q;
          reasons[|tr|] ← f;
        end
      end
    until a unchanged;
    if a precisely represents a set of models then return SAT;
    d ← adecide(a);
    if a ⊑ d then return UNKNOWN;
    a ← a ⊓ d;
    tr ← tr · d;
    reasons[|tr|] ← nil;
```

Algorithm 6: Abstract model search

Once a fixed point is reached, the result is checked to see if it precisely represents a set of models. Such a check is typically simple to implement. (In terms of abstract interpretation this step corresponds to a γ-completeness check, see [30] for details.) For example, in the propositional case one may check whether the current partial assignment assigns all variables, or alternatively, whether it satisfies at least one literal in each clause.

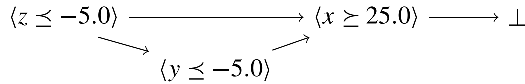
If the result of fixed point computation is neither a conflict nor a witness of satisfiability, then the abstract element is refined using an abstract decision by calling adecide (a). The resulting decision element d must be a meet irreducible that may be heuristically chosen. If the decision element fails to refine the current element, then we return an unknown result, since we have been unable to establish whether the instance is SAT or UNSAT.

5 Learning in abstract lattices

We now present our lifting of propositional conflict analysis to abstract lattices. In CDCL, the trail implicitly encodes a graph structure. The edge information is contained in the clauses

associated with each element via the *reasons* array. The FIRST-UIP algorithm [70] shown in Algorithm 4 computes a set of nodes of this graph, called a *cut*, that suffices to produce a conflict. Naively lifting the algorithm is insufficient to learn good reasons in the interval abstraction as the following example will illustrate.

Example 3 Consider the FPA formula $z = y \wedge x = y \cdot z \wedge x < 0$ and the interval assignment $\sigma = \langle z \preceq -5.0 \rangle$. Starting from σ, we can make the following deductions.



Arrows indicate sufficient conditions for deduction, e.g., $\langle x \succeq 25.0 \rangle$ can be deduced from the conjunction of $\langle z \preceq -5.0 \rangle$ and $\langle y \preceq -5.0 \rangle$. The last deduction $\langle x \succeq 25.0 \rangle$ conflicts with the constraint $x < 0$. A classic conflict cutting algorithm may analyse the above graph to conclude that $\pi = \langle z \preceq -5.0 \rangle$ is the reason for the conflict. It is easy to see though that there is a much more general reason: The conflict can be deduced in this way whenever z is negative.

5.1 Abductive reasoning and heuristic choice

A central insight is that conflict analysis performs a form of abductive reasoning: in each iteration of the conflict analysis loop, a singleton assignment a in the conflict reason is replaced by a partial assignment that is sufficient to infer a. In terms of abstract satisfaction, abduction corresponds to underapproximation of the conflict transformer [28, 30].

Example 4 Consider a clause $p \vee q$ and a partial assignment $\{q \mapsto \mathsf{t}\}$. In following concrete application of $\mathit{confs}_{p\vee q}$, we write $(p, q \mapsto v_1, v_2)$ to denote the function that maps p to v_1 and q to v_2.

$$\mathit{confs}_{p\vee q}\big(\gamma\big(\{q \mapsto \mathsf{t}\}\big)\big) = \big\{(p, q \mapsto \mathsf{f}, \mathsf{f}), (p, q \mapsto \mathsf{f}, \mathsf{t}), (p, q \mapsto \mathsf{t}, \mathsf{t})\big\}$$

Above, $\mathit{confs}_{p\vee q}$ returns the set of propositional structures that dissatisfy the formula or are approximated by $\{q \mapsto \mathsf{t}\}$. Informally, $\mathit{confs}_{p\vee q}(\gamma(\{q \mapsto \mathsf{t}\}))$ computes the most general set of circumstances under which the formula implies the truth of $q \mapsto \mathsf{t}$.

A SAT solver may decide during conflict analysis to replace the partial assignment $\{q \mapsto \mathsf{t}\}$ with $\{p \mapsto \mathsf{f}\}$ in the conflict reason. This may be modelled as application of a transformer over partial assignments:

$$\mathit{aconfs}_{p\vee q}\big(\{q \mapsto \mathsf{t}\}\big) = \{p \mapsto \mathsf{f}\}$$

Note that $\mathit{aconfs}_{p\vee q}$ underapproximates confs_{φ}, since the result of applying it does not represent the case $(p, q \mapsto \mathsf{t}, \mathsf{t})$ that is covered by the concrete computation.

5.1.1 Abductive generalisation

Propositional conflict analysis with FIRST-UIP uses only propositional abductive reasoning. In order to adapt the algorithm to perform domain-specific conflict analysis the use of a separate abduction transformer is necessary. The result of abduction should generalise the original conflict, to avoid its reexploration, and therefore the possibility of cycles. We define an abductive generalisation transformer that has this property.

Definition 6 An *abductive generalisation transformer* for a formula φ over an overapproximating abstract domain A is a function $aconfs_\varphi : A \times A \to A$, such that the properties (i) and (ii) below hold for all $r, a \in A$ where $mods_\varphi(\gamma(r)) \subseteq \gamma(a)$.

(i) $aconfs_\varphi(r, a) \sqsupseteq r$
(i) $\gamma(aconfs_\varphi(r, a)) \subseteq confs_\varphi(\gamma(a))$

The definition above requires some explanation. In conditions (i) and (ii) the element r represents a semantically expressed reason for a. More formally, we require that every model of φ represented by r is also represented by a. The result of calling $aconfs_\varphi(r, a)$ is then an element that generalises r and is still a reason for a.

Example 5 Consider the formula $x = -y$. Then $\langle x \succeq 0.0\rangle$ is a sufficient reason to determine that $\langle y \preceq 23.5\rangle$ using an appropriate deduction transformer. We use abductive generalisation to find a more general reason:

$$aconfs_{x=-y}\big(\langle x \succeq 0.0\rangle, \langle y \preceq 23.5\rangle\big) = \langle x \succeq -23.5\rangle$$

In some domains, it may be difficult to find good abductive generalisation transformers. Note that the function $(r, a) \mapsto r$ (that is, no generalisation) is always a sound choice and is sufficient for implementing ACDCL. The use of generalisation is therefore an optional opportunity to increase performance rather than a strict requirement.

The better the results of generalisation, the more powerful the results of learning. Good generalisation can be expensive though; while powerful abductive generalisation techniques may reduce the overall number of iterations of the ACDCL algorithm, the runtime required for each individual iteration may increase. As we shall see in Sect. 6, it is important to strike a careful balance to maintain overall performance.

5.1.2 Heuristic choice

Galois-connection based abstract domains have a property called *best abstraction*: In an overapproximate abstraction, concrete elements, have unique, maximally precise overapproximate representations. Similarly, every concrete transformer is overapproximated by a unique, maximally precise abstract transformer called the *best abstract transformer*. The abductive generalisation transformer does not have this property, since it is an underapproximate transformer over an overapproximate abstraction. In practice, this means that there may be multiple incomparable choices when attempting to generalise an element.

Example 6 Consider the formula φ given by $x = y + z$, and the interval elements $a = \langle x \preceq 10.0\rangle$ and $r = \langle y \preceq 0.0, z \preceq 0.0\rangle$. The element r is a sufficient reason for a, and we may apply abductive generalisation in multiple, mutually incomparable ways.

$$aconfs^1_\varphi(r, a) = \langle y \preceq 10.0, z \preceq 0.0\rangle \qquad aconfs^2_\varphi(r, a) = \langle y \preceq 0.0, z \preceq 10.0\rangle$$

Both of the abductive generalisation transformers above are sound, but they return incomparable results since neither reason is weaker or stronger than the other one. Moreover, the join of the two reasons $\langle y \preceq 10.0, z \preceq 10.0\rangle$ is not a reason for a itself, because it allows, for example, that $x = 20.0$.

In propositional conflict analysis, the lack of best conflict analysis is reflected in the cut heuristic used to extract the conflict reason. Different heuristics may produce distinct, incomparable conflict reasons.

In ACDCL, heuristic choice between reasons plays a role during abductive generalisation. As indicated in the example above, multiple reasons may be available. Abductive generalisation may choose among them based on heuristic considerations such as the state of the solver or the history of the search. We will discuss an example of an abductive generalisation heuristic in Sect. 6.

5.2 Abstract FirstUIP

```
aAnalyse(tr, reasons)
  m ← {1 ↦ ⊤, ..., |tr| ↦ ⊤};
  r ← aconfs_φ^{reasons[⊥]}(⊓tr, ⊥);
  updateMarking(m, tr, |tr|, r);
  for (i ← |tr|; UIPreached(m, tr); i ← i − 1) do
    a ← m[i];
    if a = ⊤ ∨ reasons[i] = nil then continue;
    m[i] ← ⊤;
    r ← aconfs_φ^{reasons[i]}(⊓_{1≤j<i} tr[j], a);
    updateMarking(m, tr, r);
  end
  R ← {m[i] | m[i] ≠ ⊤};
  return Unit_R;

updateMarking(m, tr, r)
  Q ← unique, subset-minimal meet decomposition of r;
  foreach q ∈ Q do
    j ← smallest index i s.t.tr[i] ⊑ q;
    m[j] ← m[j] ⊓ q;
  end
```

Algorithm 7: Abstract FIRST-UIP conflict analysis

We present our lifting of FIRST-UIP to lattices in Algorithm 7. It takes as input a trail *tr*, a reason array (which is required to contain a mapping for $\bot$). Furthermore, we assume that for each abstract model transformer f, we have a corresponding abductive generalisation $aconfs_{\varphi}^{f}$. We also assume that all abstract elements (except $\bot$) have a unique, subset-minimal meet decomposition. The main data structure is a *marking* m which maps trail indices to meet irreducible elements or $\top$. This is similar to implementations of propositional conflict analysis: There, propositions receive binary markings to indicate that they are necessary to derive the conflict. The abstract markings we use instead store for each trail element a generalisation, such that a conflict may still be derived. Initially, m maps all elements to $\top$. The procedure steps backwards through the trail, and replaces trail markings using

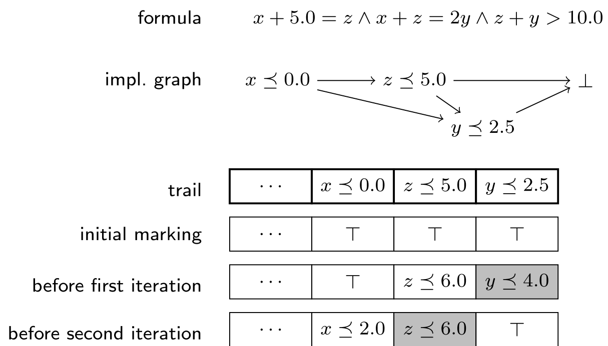


Fig. 3 Markings in abstract FIRST-UIP

reasons generated from abductive generalisation. The results of the generalisation step are decomposed into meet irreducibles and added to the marking.

At the end, a transformer $Unit_R$ is returned from the final conflict reason R. We will discuss the construction of this transformer in the next section.

An example execution of the algorithm is illustrated in Fig. 3. There, an implication graph and corresponding trail is shown which records consequences of a decision $x \preceq 0.0$. Similar to propositional CDCL, no explicit graph is constructed. Instead, the algorithm implicitly explores the graph via markings, which overapproximate the trail pointwise and encode sufficient conditions for unsatisfiability. First the algorithm determines that $\bot$ can be ensured whenever $z \preceq 6.0$ and $y \preceq 4.0$ are the case. In the first iteration, it finds that $y \preceq 4.0$ can be dropped from the reason if $x \preceq 2.0$ holds in addition to $z \preceq 6.0$.

It is an invariant during the run of the procedure that the greatest lower bound over all markings is sufficient to ensure a conflict. Hence the procedure could essentially terminate during any iteration and yield a sound global abduction result. We use the usual FIRST-UIP termination criterion and return once the number of open paths reaches 1. This number is defined as the number of indices j greater or equal to the index of the most recent decision, such that $m[j] \neq \top$.

5.3 Abstract clause learning

Propositional solvers learn new clauses that express the negation of the conflict analysis result. The new clauses open up further possibilities for deduction using the unit rule. We model learning directly as learning of a new deduction rule, rather than learning a formula in the logic. A lattice-theoretic generalisation of the unit rule is given below. Note that we define the rule directly in terms of a set of conflicting meet irreducibles, rather than their negation.

Definition 7 For an abstraction A of $\wp(Structs)$ with complementable meet irreducibles, let $R \subseteq A$ be a set of meet irreducibles such that $\sqcap R$ represents no models of φ. The *abstract unit rule* $Unit_R : A \to A$ is defined as follows.

$$Unit_R(a) \,\hat{=}\, \begin{cases} \bot & \text{if } a \sqsubseteq \sqcap R \\ \overline{r} & \text{otherwise, if } r \in R \text{ and} \\ & \forall r' \in R \setminus \{r\}.\ a \sqsubseteq r' \\ \top & \text{otherwise} \end{cases}$$

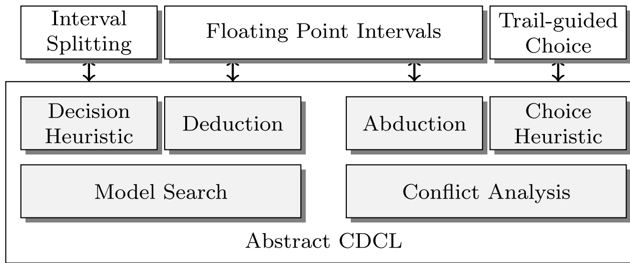


Fig. 4 FP-ACDCL solver architecture

Example 7 Let $c = \langle x{:}[0.0, 10.0], y \preceq 3.2\rangle$ be a conflicting element of φ with decomposition $R = \{\langle x \succeq 0.0\rangle, \langle x \preceq 10.0\rangle, \langle y \preceq 3.2\rangle\}$. Let $a = \langle x{:}[3.0, 4.0], y{:}[1.0, 1.0]\rangle$, then $Unit_c(a) = \bot$, since $a \sqsubseteq c$. Let $a' = \langle x{:}[3.0, 4.0]\rangle$, then $Unit_c(a') = \langle y \succ 3.2\rangle$, since $a' \sqsubseteq \langle x \succeq 0.0\rangle$ and $a' \sqsubseteq \langle x \preceq 10.0\rangle$.

The unit rule $Unit_R$ for a conflicting set of meet irreducibles R soundly overapproximates the model transformer [30]. Furthermore, it is a meet irreducible deduction transformer, so we may add it to the set of transformers F used during model search.

6 Implementation and experiments

We have implemented our approach over floating-point intervals inside the MATHSAT5 SMT solver [16]. We call our prototype tool FP-ACDCL. The implementation uses the MATHSAT5 infrastructure, but is independent of its DPLL(T) framework (although it may be used as a theory solver inside DPLL(T) if required). The implementation provides a generic, abstract CDCL framework with FIRST-UIP learning. The overall architecture is shown in Fig. 4. An instantiation requires abstraction-specific implementations of the components described earlier, including deduction, decision making and abduction with heuristic choice. We first elaborate on those aspects of the implementation and then report experimental results.

6.1 Implementation details

6.1.1 Deductions

We implement deduction using standard Interval Constraint Propagation (ICP) techniques for floating-point numbers, defined e.g., in [8, 54]. The implementation operates on CNF formulae over floating-point predicates.

Propagation is performed using an occurrence-list approach, which associates with each variable a list of the FPA clauses in which the variable occurs. Learnt unit transformers are stored as vectors of meet irreducible elements and are propagated in a similar way. When a deduction is made, we scan the list of affected clauses to check for new deductions to be added to the trail. This is done by applying ICP projection functions to the floating-point predicates in a way that combines purely propositional with theory-specific reasoning. A predicate is conflicting if some variable is assigned the empty interval during ICP. If all predicates of a clause are contradicting, then we have found a conflict with the current interval assignment and *ded* returns $\bot$. If all but one predicate in a clause are conflicting,

then the result of applying ICP to the remaining predicate is the deduction result. In this case, *ded* returns a list containing one meet irreducible $\langle x \succeq b \rangle$ (or $\langle x \preceq b \rangle$) for each new bound inferred.

6.1.2 Decisions

FP-ACDCL performs decisions by adding to the trail one meet irreducible element $\langle x \succeq b \rangle$ or $\langle x \preceq b \rangle$ that does not contradict the previous value of x. Clearly, there are many possible choices for (i) selecting a variable x, (ii) selecting a bound b, and (iii) choosing between $\langle x \succeq b \rangle$ and $\langle x \preceq b \rangle$.

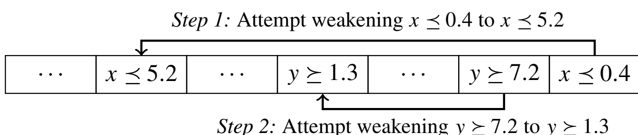
In propositional CDCL, each variable can be assigned at most once. In our lifting, a variable can be assigned multiple times with increasingly precise bounds. We have found some level of *fairness* to be critical for performance. Decisions should be balanced across different variables and upper and lower bounds. A strategy that proceeds in a "depth-first" manner, in which the same variable is refined using decisions until it has a singleton value, shows inferior performance compared to a "breadth-first" exploration, in which intervals of all the variables are restricted uniformly. We interpret this finding as indication that the value of abstraction lies in the fact that the search can be guided effectively using general, high-level reasoning, before considering very specific cases.

FP-ACDCL currently performs decisions as follows: (i) variables are statically ordered, and the selection on which variable x to branch is cyclic across this order; (ii) the bound b is chosen to be an approximation of the arithmetic average between the current bounds l and u on x; note that the arithmetic average is different from the median, since floating-point values are non-uniformly distributed across the reals; (iii) the choice between $\langle x \succeq b \rangle$ and $\langle x \preceq b \rangle$ is random. Considering the advances in heuristics for propositional SAT, there is likely a lot of room for enhancing this. In particular, the integration of fairness considerations with activity-based heuristics typically used in modern CDCL solvers could lead to similar performance improvements. This is part of ongoing and future work.

6.1.3 Generalised explanations for conflict analysis

In abduction, a trade-off must be made between finding reasons quickly and finding very general reasons. We perform abduction that relaxes bounds iteratively. As mentioned earlier, there may be many incomparable relaxations. Our experiments suggest that the way in which bounds are relaxed is extremely important for performance. Fairness considerations similar to those mentioned for the decision heuristic need to be taken into account. However, there is an additional, important criterion. Learnt unit rules are used to drive backjumping. It is therefore preferable to learn deduction rules that allow for backjumping higher in the trail. This will lead to propagations that are affected by a smaller number of decisions, and thus will hold for a larger portion of the search space.

Our choice heuristic, called *trail-guided choice*, is abstraction-independent, and is both fair and aims to increase backjump potential. In the first step, we remove all bounds over variables from the initial reason q which are irrelevant to the deduction. Then we step backwards through the trail and attempt to weaken the current element q using trail elements. The process is illustrated below.



When an element tr_j is encountered such that tr_j is a bound on a variable x that is used in q (that is, $q \sqsubseteq tr_j$), the first thing we do is to attempt to weaken q by replacing the bound tr_j with the most recent trail element more general than tr_j. If no such element exists, we replace tr_j with the trivial bound $\langle x \succeq \min_{\preceq}(\mathbb{F})\rangle$. We check whether the weakened q is still sufficiently strong to deduce d. If so, we set q as the candidate generalisation and continue stepping backwards through the trail by processing the next element tr_{j-1}. If not, we undo the weakening, and try to consider intermediate cases, that is elements weaker than q but stronger than tr_j, but only performing a *bounded* number of attempts on the current variable x. For example, if q contains $x \in [l : u]$ and tr_j is $\langle x \preceq c\rangle$ with $u \preceq c$, we try setting the interval for x to $[l : u + (c - u)/2]$ and so on, until either no further generalisation on the upper bound of x is possible, or we reach the limit on the number of attempts. The algorithm then terminates once no further generalisations are possible.

Since we step backwards in order of deduction, we heuristically increase the potential for backjumps: the procedure never weakens a bound that was introduced early during model search at the expense of having to uphold a bound that is ensured only at a deep level of the search.

We have experimented with stronger but computationally more expensive generalisation techniques such as finding maximal bounds for deductions by search over floating-point values, as well as with different limits on the number of generalisation attempts on a single bound. Our experiments, reported in Sect. 6.2.2, indicate that the cheaper technique described above is more effective overall, and that using a small cutoff value gives the best trade-off between cost and quality of the generalisations. However, we believe that more sophisticated strategies might provide further benefits. In particular, we see two main avenues for improvement: first, for many deductions it is possible to implement good or optimal abduction transformers effectively without search. Second, we expect that dynamic heuristics that take into account statistical information may guide conflict analysis towards useful clauses.

6.2 Experimental evaluation

We have evaluated our prototype FP-ACDCL tool over a set of 213 benchmark formulas, both satisfiable and unsatisfiable. The formulas have been generated from problems that check

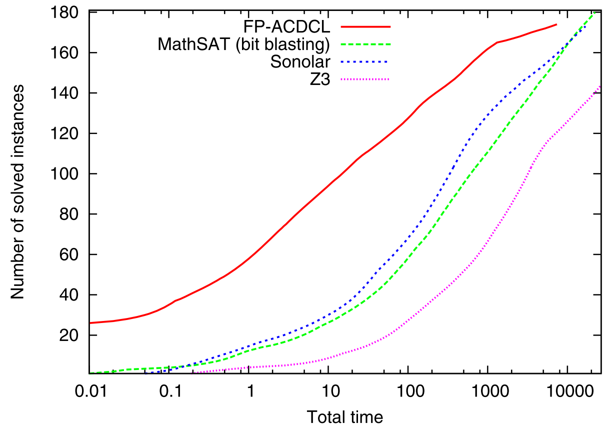
1. ranges on numerical variables and expressions,
2. error bounds on numerical computations using different orders of evaluation of subexpressions, and
3. feasibility of systems of inequalities over bounded floating-point variables.

The first two sets originate from verification problems on C programs performing numerical computations, whereas the instances in the third set are randomly generated. Our benchmarks and the FP-ACDCL tool are available for reproducing the experiments at http://es.fbk.eu/people/griggio/papers/FMSD-fmcad12.tar.bz2. All results have been obtained on an Intel Xeon machine with 2.6 GHz and 16 GB of memory running Linux, with a time limit of 1200 seconds.

6.2.1 Comparison with bit-vector encodings

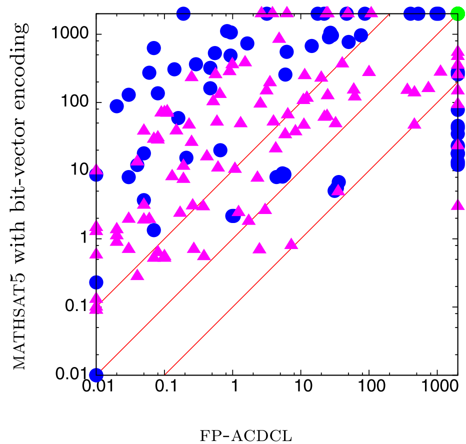
In the first set of experiments, we have compared FP-ACDCL with the current state-of-the-art procedures for floating-point arithmetic based on encoding into bit-vectors. For this, we have compared against all the SMT solvers supporting FPA that we were aware of, namely

Fig. 5 Comparison of FP-ACDCL against various SMT solvers using bit-vector encoding of floating-point operations



Solver	Num. instances	Total time
MATHSAT5	181	22271
FP-ACDCL	175	7409
SONOLAR	174	16926
Z3	145	26522

Fig. 6 Detailed comparison of FP-ACDCL against MATHSAT5 using bit-vector encoding of floating-point operations. *Circles* indicate unsatisfiable instances, triangles satisfiable ones. *Points on the borders* indicate timeouts (1200 s)



Z3 [26], SONOLAR [50] and MATHSAT5 [16]. All three solvers use a bit-vector encoding of floating-point arithmetic which is then solved via reduction to SAT(bit-blasting). For each tool, we used the default options. The results of this comparison are reported in Figs. 5 and 6. The plot in Fig. 5 shows the number of succesfully solved instances for each system (on the Y axis) and the total time needed for solving them (on the X axis). From it, we can see that FP-ACDCL clearly outperforms Z3 both in number of instances and in total execution time, and that FP-ACDCL solves one instance more than SONOLAR, but it is overall significantly faster. Compared to MATHSAT5, the results are mixed: as can be seen in the scatter plot of Fig. 6, on one hand FP-ACDCL is much faster than MATHSAT5 in the majority of instances that both tools can solve, but on the other hand MATHSAT5 seems to still have an advantage in terms of scalability, solving overall 6 more instances than FP-ACDCL. More generally, there are some instances that turn out to be relatively easy for solvers based on bit-blasting, but cannot be solved by FP-ACDCL. This is not surprising, since there are simple instances that are not amenable to analysis with ICP, even with the addition of decision-making and

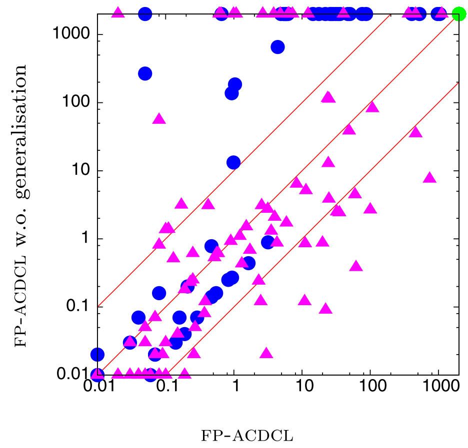


Fig. 7 Effects of generalisations in conflict analysis. *Circles* indicate unsatisfiable instances, *triangles* satisfiable ones. *Points on the borders* indicate timeouts (1200 s)

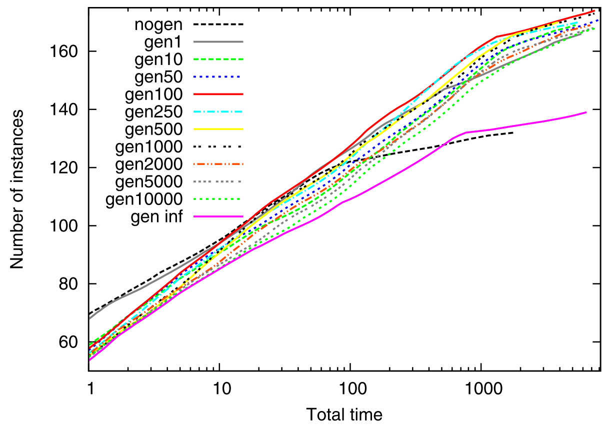


Fig. 8 Comparison of different strategies for conflict generalisation in FP-ACDCL

Configuration	Num. instances	Total time
gen100	175	7409
gen1000	174	7348
gen50	172	8181
gen500	171	3876
gen250	171	5576
gen10	170	5263
gen2000	170	6827
gen5000	169	7204
gen10000	169	7574
gen1	167	5842
gen inf	140	6390
nogen	133	1783

learning.[1] To handle such cases, our framework can be instantiated with abstract domains or combinations of domains [21] that are better suited to the problems under analysis. Moreover, bit-blasting approaches can take advantage of highly efficient SAT solvers, which are the result of years of development, optimizations and fine tuning, whereas our FP-ACDCL tool should still be considered a prototype.

[1] A simple example of this is the formula $x = y \wedge x \neq y$, which requires an abstraction that can express relationships between variables. Intervals are insufficient to efficiently solve this problem.

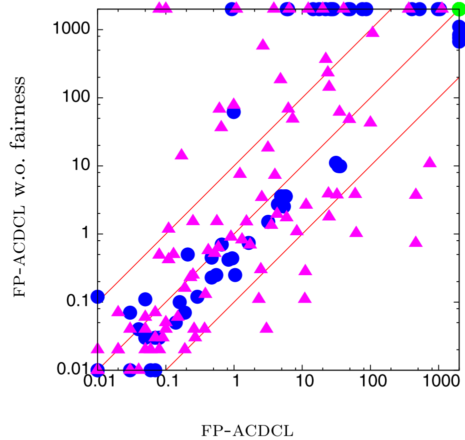


Fig. 9 Effects of fairness in branching heuristic. *Circles* indicate unsatisfiable instances, *triangles* satisfiable ones. *Points on the borders* indicate timeouts (1200 s)

6.2.2 Impact of optimizations

The second set of experiments is aimed at evaluating the impact of our variable selection and generalisation techniques. In order to evaluate our novel generalisation technique, we have first run FP-ACDCL with generalisation of deductions turned off, and compared it with the default FP-ACDCL. Essentially, FP-ACDCL without generalisation corresponds to a naive lifting of the conflict analysis algorithm. The results are summarised in Fig. 7. From the plot, we can clearly see that generalisation is crucial for the performance of FP-ACDCL: without it, the tool times out in 42 more cases, whereas there is no instance that can be solved only without generalisation. However, there are a number of instances for which performance degrades when using generalisations, sometimes significantly. This can be explained by observing that (i) generalisations come at a runtime cost, which can sometimes induce a non-negligible overhead; (ii) the performance degradation occurs on satisfiable instances (shown in a lighter colour in the plots), for which it is known that the behaviour of CDCL-based approaches is typically unstable (even in the propositional case).

Subsequently, we have performed a more in-depth evaluation of the effects of using different generalisation strategies. In particular, we have compared different versions of FP-ACDCL with different cutoff values for the number of generalisation attempts in the trail-guided procedure described in Sect. 6.1.3. The results are collected in Fig. 8. In the figure, ‘genX’ stands for a configuration with a cutoff value of ‘X’ generalisation attempts, ‘nogen’ is the configuration where no generalisation is performed, and ‘gen inf’ in the one which does not impose any bound on the number of generalisation attempts. From the plot, we can see that the strategies that impose a limit to the number of generalisation attempts significantly outperform both the unconstrained strategy and the naive one that uses no generalisation at all. The results also indicate that there is a trade-off between the quality of the generalisation and the cost of performing it, with a ‘sweet spot’ (for our benchmarks) reached using a cutoff value of 100 attempts per step.

Finally, we have performed a further set of experiments in order to evaluate the impact of fairness in the variable selection herustics for branching and conflict generalisation. We have compared the default version of FP-ACDCL (which tries to ensure fairness as described in Sect. 6.1.2 and Sect. 6.1.3) with a version in which variables are selected randomly. The results, shown in Fig. 9, demonstrate that fairness is a very important factor for the performance of FP-ACDCL: while there are instances which are solved more efficiently when

selecting variables randomly,[2] overall the use of fair selection strategies allows FP-ACDCL to solve 23 more instances.

7 A survey of related work

The work presented in this paper may be understood in the context of efforts to unify SAT techniques and abstract interpretation: the work in [29] describes an abstract interpreter for programs that uses SAT-style conflict-driven learning; [28] and [9] show, respectively, that DPLL-based SAT solvers and DPLL(T)-based SMT solvers are abstract interpreters; the ACDCL algorithm [30] is presented in ACDCL. In [10], we give an interpolation procedure for some instances of ACDCL, including FP-ACDCL.

We now separately survey work in three related branches of research: (1) the analysis of floating-point computations, (2) lifting existing decision procedure architectures to richer problem domains and (3) automatic and intelligent precision refinement of abstract analyses.

7.1 Reasoning about floating-point numbers

This section briefly surveys work in interactive theorem proving, abstract interpretation and decision procedures that target floating-point problems. For a discussion of the special difficulties that arise in this area, see [59].

7.1.1 Theorem proving

Various floating-point axiomatisations and libraries for interactive theorem provers exist [24, 40, 53, 57]. Theorem provers have been applied extensively to proving properties over floating-point algorithms or hardware [1, 41–44, 49, 60, 64]. While theorem proving approaches have the potential to be sound and complete, they require substantial manual work, although sophisticated (but incomplete) strategies exist to automate substeps of the proof, e.g., [2]. A preliminary attempt to integrate such techniques with SMT solvers has recently been proposed in [18].

7.1.2 Abstract interpretation

Analysis of floating-point computations has also been extensively studied in abstract interpretation. An approach to specifying floating-point properties over programs was proposed in [7]. A number of general purpose abstract domains have been constructed for the analysis of floating-point programs [12–15, 46, 56]. In addition, specialised approaches exist which target specific problem domains such as digital filters [32, 58]. The approaches discussed so far mainly aim at establishing the result of a floating-point computation. An orthogonal line of research is to analyse the deviation of a floating-point computation from its real counterpart by studying the propagation of rounding errors [35, 37]. Case studies for this approach are given in [27, 38]. Abstract interpretation techniques provide a soundness guarantee, but may yield imprecise results.

[2] As can be seen from Fig. 9, this happens especially on satisfiable formulas, for which the stability considerations done above when evaluating the benefits of generalisation still apply.

7.1.3 Decision procedures

In the area of decision procedures, study of floating-point problems is relatively scarce. Work in constraint programming [55] shows how approximation with real numbers can be used to soundly restrict the scope of floating-point values. In [8], a symbolic execution approach for floating-point problems is presented, which combines interval propagation with explicit search for satisfiable floating-point assignments. An SMTLIB theory of FPA was presented in [63]. Recent decision procedures for floating-point logic are based on propositional encodings of floating-point constraints. Examples of this approach are implemented in MATHSAT5 [16], CBMC [17] and Sonolar [45]. A difficulty of this approach is that even simple floating-point formulas can have extremely large propositional encodings, which can be hard for current SAT solvers. This problem is addressed in [11], which uses a combination of over- and underapproximate propositional abstractions in order to keep the size of the search space as small as possible.

7.2 Lifting decision procedures

The practical success of CDCL solvers has given rise to various attempts to lift the algorithmic core of CDCL to new problem domains. This idea is extensively studied in the field of satisfiability modulo theories. The most popular such lifting is the DPLL(T) framework [34], which separates theory-specific reasoning from Boolean reasoning over the structure of the formula. Typically a propositional CDCL solver is used to reason about the Boolean structure while an ad-hoc procedure is used for theory reasoning. The DPLL(T) framework can suffer from some difficulties that arise from this separation. To alleviate these problems, approaches such as *theory decisions on demand* [4] and theory-based decision heuristics [36] have been proposed.

Our work is co-located in the context of natural-domain SMT [19], which aims to lift steps of the CDCL algorithm to operate directly over the theory. Notable examples of such approaches have been presented for equality logic with uninterpreted functions [3], linear real arithmetic and difference logic [19, 51], linear integer arithmetic [47], nonlinear integer arithmetic [33], and nonlinear real arithmetic [48]. The work in [33] is most similar to ours since it also operates over intervals and uses an implication graph construction.

We follow a slightly different approach to generalisation based on abstract interpretation. The work in [28] shows that SAT solvers can naturally be considered as abstract interpreters for logical formulas. Generalisations can then be obtained by using different abstract domains. Our work is an application of this insight. A similar line of research was independently undertaken in [68, 69], which presents an abstract-interpretation based generalisation of Stålmarck's method and an application to computation of abstract transformers.

7.3 Refining abstract analyses

A number of program analyses exist that use decision procedures or decision procedure architectures to refine a base analysis. A lifting of CDCL to program analyses over abstract domains is given in [29]. In [52], a decision-procedure based software model checker is presented that imitates the architecture of a CDCL solver. A lifting of DPLL(T) to refinement of abstract analyses is presented in [39] which combines a CDCL solver with an abstract interpreter.

Modern CDCL solvers can be viewed as refinements of the original DPLL algorithm [25], which is based on case-analysis. Case analysis has been studied in the abstract interpretation

literature. The formal basis is given by cardinal power domains, already discussed in [21], in which a base domain is refined with a lattice of cases. The framework of *trace partitioning* [62] describes a systematic refinement framework for programs based on case analysis. The DPLL algorithm can be viewed as a special instance of dynamic trace partitioning applied to the analysis of logical formulas.

8 Conclusions and future work

We have presented a decision procedure for the theory of floating-point arithmetic based on a strict lifting of the conflict analysis algorithm used in modern CDCL solvers to abstract domains. We have shown that, for a certain class of formulas, this approach significantly outperforms current complete solvers based on bit-vector encodings. Both our formalism and our implementation are modular and separate the CDCL algorithm from the details of the underlying abstraction. Furthermore, the overall architecture is not tied to analysing properties over floating-point formulas.

We are interested in a number of avenues of future research. One of these is a comparison of abstract CDCL and DPLL(T)-based architectures, and investigating possible integrations. Another avenue of research is instantiating ACDCL with richer abstractions (e.g., octagons). Combination and refinements of abstractions are well studied in the abstract interpretation literature [21]. Recent work [22] has shown that Nelson-Oppen theory combination is an instance of a product construction over abstract domains. We hope to apply this work to obtain effective theory combination within ACDCL. In addition, product constructions can be used to enhance the reasoning capabilities within a single theory, e.g., by fusing interval-based reasoning over floating-point numbers and propositional reasoning about the corresponding bit-vector encoding.

We see this work as a step towards integrating the abstract interpretation point of view with algorithmic advances made in the area of decision procedures. Black-box frameworks such as DPLL(T) abstract away from the details of their component procedures. Abstract interpretation can be used to express an orthogonal, algebraic "white-box" view which, we believe, has uses in both theory and practice.

Open Access This article is distributed under the terms of the Creative Commons Attribution License which permits any use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

References

1. Akbarpour B, Abdel-Hamid A, Tahar S, Harrison J (2010) Verifying a synthesized implementation of IEEE-754 floating-point exponential function using HOL. Comput J 53(4):465–488
2. Ayad A, Marché C (2010) Multi-prover verification of floating-point programs. In: Proc of international joint conference on automated reasoning. Springer, Berlin, pp 127–141
3. Badban B, van de Pol J, Tveretina O, Zantema H (2007) Generalizing DPLL and satisfiability for equalities. Inf Comput 205(8):1188–1211
4. Barrett C, Nieuwenhuis R, Oliveras A, Tinelli C (2006) Splitting on demand in SAT modulo theories. In: Proc of logic programming, artificial intelligence and reasoning, pp 512–526
5. Barrett C, Sebastiani R, Seshia SA, Tinelli C (2009) Satisfiability modulo theories. In: Handbook of satisfiability. IOS Press, Amsterdam, pp 825–885
6. Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X (2003) A static analyzer for large safety-critical software. In: Proc of programming language design and implementation. ACM, New York, pp 196–207

7. Boldo S, Filliâtre J (2007) Formal verification of floating-point programs. In: Proc of computer arithmetic. IEEE, New York, pp 187–194
8. Botella B, Gotlieb A, Michel C (2006) Symbolic execution of floating-point computations. Softw Test Verif Reliab 16(2):97–121
9. Brain M, D'Silva V, Haller L, Griggio A, Kroening D (2013) An abstract interpretation of DPLL(T). In: Proc of verification, model checking and abstract interpretation. Springer, Berlin, pp 455–475
10. Brain M, D'Silva V, Haller L, Griggio A, Kroening D (2013) Interpolation-based verification of floating-point programs with abstract CDCL. In: Proc of static analysis symposium. Springer, Berlin, pp 412–432
11. Brillout A, Kroening D, Wahl T (2009) Mixed abstractions for floating-point arithmetic. In: Proc of formal methods in computer-aided design. IEEE, New York, pp 69–76
12. Chapoutot A (2010) Interval slopes as a numerical abstract domain for floating-point variables. In: Proc of static analysis symposium. Springer, Berlin, pp 184–200
13. Chen L, Miné A, Cousot P (2008) A sound floating-point polyhedra abstract domain. In: Proc of Asian symposium on programming languages. Springer, Berlin, pp 3–18
14. Chen L, Miné A, Wang J, Cousot P (2009) Interval polyhedra: an abstract domain to infer interval linear relationships. In: Proc of static analysis symposium. Springer, Berlin, pp 309–325
15. Chen L, Miné A, Wang J, Cousot P (2010) An abstract domain to discover interval linear equalities. In: Proc of verification, model checking and abstract interpretation. Springer, Berlin, pp 112–128
16. Cimatti A, Griggio A, Schaafsma B, Sebastiani R (2013) The MathSAT5 SMT solver. In: Proc of tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 93–107
17. Clarke E, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In: Proc of tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 168–176
18. Conchon S, Melquiond G, Roux C, Iguernelala M (2012) Built-in treatment of an axiomatic floating-point theory for SMT solvers. In: SMT workshop
19. Cotton S (2010) Natural domain SMT: a preliminary assessment. In: Proc of formal modeling and analysis of timed systems. Springer, Berlin, pp 77–91
20. Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc of principles of programming languages. ACM Press, New York, pp 238–252
21. Cousot P, Cousot R (1979) Systematic design of program analysis frameworks. In: Proc of principles of programming languages. ACM Press, New York, pp 269–282
22. Cousot P, Cousot R, Mauborgne L (2011) The reduced product of abstract domains and the combination of decision procedures. In: Proc of foundations of software science and computation structures. Springer, Berlin, pp 456–472
23. Cowlishaw M (ed) (2008) IEEE standard for floating-point arithmetic. IEEE, New York, pp 1132–1138
24. Daumas M, Rideau L, Théry L (2001) A generic library for floating-point numbers and its application to exact computing. In: Proc of theorem proving in higher order logics. Springer, Berlin, pp 169–184
25. Davis M, Logemann G, Loveland D (1962) A machine program for theorem-proving. Commun ACM 5:394–397
26. de Moura L, Bjørner N (2008) Z3: an efficient SMT solver. In: Proc of tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 337–340
27. Delmas D, Goubault E, Putot S, Souyris J, Tekkal K, Védrine F (2009) Towards an industrial use of FLUCTUAT on safety-critical avionics software. In: Proc of formal methods for industrial critical systems. Springer, Berlin, pp 53–69
28. D'Silva V, Haller L, Kroening D (2012) Satisfiability solvers are static analyzers. In: Proc of static analysis symposium. Springer, Berlin, pp 317–333
29. D'Silva V, Haller L, Kroening D, Tautschnig M (2012) Numeric bounds analysis with conflict-driven learning. In: Proc of tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 48–63
30. D'Silva V, Haller L, Kroening D (2013) Abstract conflict driven learning. In: Proc of principles of programming languages. ACM Press, New York, pp 143–154
31. D'Silva V, Haller L, Kroening D (2014) Abstract satisfaction. In: Proc of principles of programming languages (to appear). ACM, New York
32. Feret J (2004) Static analysis of digital filters. In: Proc of European symposium on programming. Springer, Berlin, pp 33–48
33. Fränzle M, Herde C, Teige T, Ratschan S, Schubert T (2007) Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. J Satisf Boolean Model Comput 1(3–4):209–236
34. Ganzinger H, Hagen G, Nieuwenhuis R, Oliveras A, Tinelli C (2004) DPLL(T): fast decision procedures. In: Proc of computer aided verification. Springer, Berlin, pp 175–188

35. Ghorbal K, Goubault E, Putot S (2009) The zonotope abstract domain Taylor1+. In: Proc of computer aided verification. Springer, Berlin, pp 627–633
36. Goldwasser D, Strichman O, Fine S (2008) A theory-based decision heuristic for DPLL(T). In: Proc of formal methods in computer-aided design. IEEE Press, New York, pp 1–8
37. Goubault E (2001) Static analyses of the precision of floating-point operations. In: Proc of static analysis symposium. Springer, Berlin, pp 234–259
38. Goubault E, Putot S, Baufreton P, Gassino J (2007) Static analysis of the accuracy in control systems: principles and experiments. In: Proc of formal methods for industrial critical systems. Springer, Berlin, pp 3–20
39. Harris WR, Sankaranarayanan S, Ivančić F, Gupta A (2010) Program analysis via satisfiability modulo path programs. In: Proc of principles of programming languages, pp 71–82
40. Harrison J (1999) A machine-checked theory of floating point arithmetic. In: Proc of theorem proving in higher order logics. Springer, Berlin, pp 113–130
41. Harrison J (2000) Floating point verification in HOL light: the exponential function. Form Methods Syst Des 16(3):271–305
42. Harrison J (2000) Formal verification of floating point trigonometric functions. In: Proc of formal methods in computer-aided design. Springer, Berlin, pp 217–233
43. Harrison J (2003) Formal verification of square root algorithms. Form Methods Syst Des 22(2):143–153
44. Harrison J (2007) Floating-point verification. J Univers Comput Sci 13(5):629–638
45. Jan Peleska EV, Lapschies F (2011) Automated test case generation with SMT-solving and abstract interpretation. In: Proc of nasa formal methods. Springer, Berlin, pp 298–312
46. Jeannet B, Miné A (2009) Apron: a library of numerical abstract domains for static analysis. In: Proc of computer aided verification. Springer, Berlin, pp 661–667
47. Jovanovic D, de Moura L (2011) Cutting to the chase: solving linear integer arithmetic. In: Proc of conference on automated deduction. Springer, Berlin, pp 338–353
48. Jovanovic D, de Moura L (2012) Solving non-linear arithmetic. In: Proc of international joint conference on automated reasoning. Springer, Berlin, pp 339–354
49. Kaivola R, Aagaard M (2000) Divider circuit verification with model checking and theorem proving. In: Proc of theorem proving in higher order logics. Springer, Berlin, pp 338–355
50. Lapschies F, Peleska J, Gorbachuk E, Mangels T (2012) SONOLAR SMT-solver. In: Satisfiability modulo theories competition 2012 system description
51. McMillan K, Kuehlmann A, Sagiv M (2009) Generalizing DPLL to richer logics. In: Proc of computer aided verification. Springer, Berlin, pp 462–476
52. McMillan KL (2010) Lazy annotation for program testing and verification. In: Proc of computer aided verification. Springer, Berlin, pp 104–118
53. Melquiond G (2012) Floating-point arithmetic in the Coq system. Inf Comput 216:14–23
54. Michel C (2002) Exact projection functions for floating point number constraints. In: Annals of mathematics and artificial intelligence
55. Michel C, Rueher M, Lebbah Y (2001) Solving constraints over floating-point numbers. In: Seventh international conference on principles and practice of constraint programming. Springer, Berlin, pp 524–538
56. Miné A (2004) Relational abstract domains for the detection of floating-point run-time errors. In: Proc of European symposium on programming. Springer, Berlin, pp 3–17
57. Miner PS (1995) Defining the IEEE-854 floating-point standard in PVS. PVS. Technical Memorandum 110167, NASA, Langley Research
58. Monniaux D (2005) Compositional analysis of floating-point linear numerical filters. In: Proc of computer aided verification. Springer, Berlin, pp 199–212
59. Monniaux D (2008) The pitfalls of verifying floating-point computations. ACM Trans Program Lang Syst 30(3)
60. Moore JS, Lynch T, Kaufmann M (1996) A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating-point division algorithm. Trans Comput 47:913–916
61. Muller JM, Brisebarre N, de Dinechin F, Jeannerod CP, Lefèvre V, Melquiond G, Revol N, Stehlé D, Torres S (2010) Handbook of floating-point arithmetic, 1st edn. Springer, Berlin
62. Rival X, Mauborgne L (2007) The trace partitioning abstract domain. ACM Trans Program Lang Syst 29(5):26
63. Rümmer P, Wahl T (2010) An SMT-LIB theory of binary floating-point arithmetic. In: SMT workshop
64. Russinoff D (1998) A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. LMS J Comput Math 1:148–200
65. Sakallah KA, Marques-Silva J (2011) Anatomy and empirical evaluation of modern SAT solvers. Bull Eur Assoc Theor Comput Sci 103:96–121

66. Silva JPM, Lynce I, Malik S (2009) Conflict-driven clause learning SAT solvers. In: Handbook of satisfiability. IOS Press, Amsterdam, pp 131–153
67. Silva JPM, Sakallah KA (1999) GRASP: a search algorithm for propositional satisfiability. Trans Comput 48(5):506–521
68. Thakur A, Reps T (2012) A generalization of Stålmarck's method. In: Proc of static analysis symposium Springer, Berlin, pp 334–351
69. Thakur A, Reps T (2012) A method for symbolic computation of abstract operations. In: Proc of computer aided verification. Springer, Berlin, pp 174–192
70. Zhang L, Madigan CF, Moskewicz MW, Malik S (2001) Efficient conflict driven learning in a Boolean satisfiability solver. In: International conference on computer-aided design. ACM, New York, pp 279–285