Abstract Conflict Driven Learning

Vijay D’Silva
Department of Computer Science
University of California, Berkeley
vijayd@eecs.berkeley.edu

Leopold Haller
Department of Computer Science
Oxford University
leopold.haller@cs.ox.ac.uk

Daniel Kroening
Department of Computer Science
Oxford University
daniel.kroening@cs.ox.ac.uk

Abstract

Modern satisfiability solvers implement an algorithm, called Conflict
Driven Clause Learning, which combines search for a model
with analysis of conflicts. We show that this algorithm can be gen-
eralised to solve the lattice-theoretic problem of determining if an
additive transformer on a Boolean lattice is always bottom. Our
generalised procedure combines overapproximations of greatest
fixed points with underapproximations of least fixed points to obtain
more precise results than computing fixed points in isolation. We
generalise implication graphs used in satisfiability solvers to derive
underapproximate transformers from overapproximate ones. Our
generalisation provides a new method for static analyzers that op-
erate over non-distributive lattices to reason about properties that
require disjunction.

Categories and Subject Descriptors F.3 [Logics and Meanings of
Programs]: Specifying and Verifying and Reasoning about Programs

Keywords Satisfiability, Conflict Driven Clause Learning, Lattices

1. The Algebraic Essence of Satisfiability Solvers

The performance of solvers for the Boolean satisfiability problem
(SAT) has improved at an exponential rate in the last decade. Several
factors contribute to these improvements including an elegant
algorithm, efficient, architecture-aware implementations of data
structures, and heuristics that exploit the non-adversarial nature of
practical problem instances. A recent survey of these developments
by Malik and Zhang [21] concludes with the following question:

“Given its theoretical hardness, the practical success of
SAT has come as a surprise to many in the computer science
community. [...] Can we take these lessons to other problems
da...
We introduce the bottom-everywhere problem and show that satis-
pointwise meet
The generalisation of implication graphs to abstract domains is
The paper is organised as follows: We introduce the bottom-
ACDCL
This paper presents
\[ \sim \]
If
Proposition 1. require the following property of De Morgan duals.
of a function
form a complete lattice under the pointwise order. The
is similarly defined. The set of transformers on a complete lattice
map into a lattice is defined as
functions from a set to a poset holds if
pointwise order
and extensive transformer, and a
lower closure
upper closure
idempotent
completely multiplicative. The function
maps
additive
if
bottom
element called
top
A lattice is called
instances of this problem. Subsequent sections lift
arguments that apply to all instantiations of ACDCL.
The paper is organised as follows: We introduce the bottom-
everywhere problem in Section 2 and discuss several instances of
this problem. We apply abstract interpretation to derive over- and
underapproximate solutions to the bottom-everywhere problem in
Section 3, and combine these approximations in Sections 4 and 5.
The generalisation of implication graphs to abstract domains is
discussed and illustrated in Section 6.

2. The Bottom-Everywhere Problem
We introduce the bottom-everywhere problem and show that satis-
fiability of formulae and error reachability in transition systems are
instances of this problem. Subsequent sections lift CDCL to solve
this problem.

Lattice-Theoretic Terminology A lattice is called bounded if it
has a greatest element, called top and denoted \( \top \), and has a least
element called bottom and denoted \( \bot \). A function \( f \) on a complete
lattice \( L \) is additive if \( f(x \cup y) = f(x) \cup f(y) \) and is completely
additive if \( f(\bigcup X) = \bigcup f(X) \). A completely additive function
maps \( \bot \). The dual notions are called multiplicative and
completely multiplicative.
A function is reductive if \( f(x) \subseteq x \) for all \( x \) and is extensive if \( f(x) \supseteq x \) for all \( x \). A function is idempotent if \( f(f(x)) = f(x) \) for all \( x \). A transformer is a
monotone function on a lattice. An upper closure is an idempotent
and extensive transformer, and a lower closure is an idempotent
and reductive transformer. The pointwise order \( f \sqsubseteq g \) between
functions from a set to a poset holds if \( f(x) \sqsubseteq g(x) \) holds for all \( x \).
The pointwise meet of \( f \) and \( g \), denoted \( f \sqcap g \), where both functions
map into a lattice is defined as \( \lambda x. f(x) \sqcap g(x) \). The pointwise join
is similarly defined. The set of transformers on a complete lattice
form a complete lattice under the pointwise order. The De Morgan
dual of a function \( f \) on a Boolean lattice is \( \overline{\sim f} \). We require the following property of De Morgan duals.

Proposition 1. If \( f \) is a completely additive, reductive function on a
powerset lattice, \( \overline{\sim f} \) is a completely multiplicative, extensive function.

The least fixed point of a transformer \( f \) on a complete lattice is
denoted lfp\((f)\) and the greatest fixed point is denoted gfp\((f)\).

2.1 Bottom and Top Everywhere
The material below is new.

Definition 2. A function \( f \) on a bounded lattice is bottom-everywhere
if \( f(x) = \bot \) for all \( x \). The bottom-everywhere problem is to deter-
mine if a function on a lattice is bottom-everywhere.

A non-bottom witness is an element \( a \) such that \( f(a) \) is not bot-
tom. A non-bottom witness \( a \) is minimal if no \( b \sqsubseteq a \) is a non-bottom
witness. The top-everywhere property and top-everywhere problem
are similarly defined. The dual notions for the top-everywhere
problem are a non-top witness and a maximal non-top witness.
In this paper, we consider the bottom-everywhere problem for
completely additive, reductive functions on powerset lattices.

Lemma 3. A completely additive, reductive function on a powerset
lattice is a lower closure.

Proof. Consider a function \( f : \mathcal{P}(S) \to \mathcal{P}(S) \). For each \( x \) in
\( S \), \( f(\{x\}) \) is either \( \emptyset \) or is \( \{x\} \), because \( f \) is reductive. Since \( f \)
is completely additive, \( f(X) = \bigcup f(X) \) for every subset \( X \) of \( S \).
In particular, \( f(X) = \{x \in X \mid f(x) \neq \emptyset\} \). It follows that
\( f(f(X)) = f(X) \).

Due to Lemma 3 we abbreviate `completely additive, reductive function'
to `additive closure` for the rest of the paper. Theorem 4 below is a consequence of Lemma 3. It is straightforward to prove
but the fixed point view is valuable because an abstraction of a lower
closure may not be idempotent, in which case iterating a transformer
in the abstract yields strictly more precision than applying it once.

Theorem 4. The following statements are equivalent for a com-
pletely additive, reductive function \( f \) on a powerset lattice.
1. \( f \) is bottom-everywhere.
2. \( gfp(f) \) is bottom.
3. \( f \) is top-everywhere.
4. \( lfp(f) \) is top.

The implication from 1 to 2 is straightforward, while the implica-
tion 2 to 1 holds because \( f \) is a lower closure. The equivalence of 1
and 3 follows by negation and of 3 and 4 due to the closure property
and complementation of fixed points.

Next, we show that two well-known problems, satisfiability and
error reachability, can be reduced to the bottom-everywhere problem.
Neither reduction is mathematically surprising but allows us to think
of decision problems rather than function problems.

2.2 Unsatisﬁability as Bottom Everywhere
Let \( \text{Struct} \) be a set of structures and \( \text{Form} \) be a set of formulae,
and \( \models \subseteq \text{Struct} \times \text{Form} \) be a satisfaction relation between
structures and formulae. If \( (\sigma, \phi) \) is in \( \models \), we write \( \sigma \models \phi \), and say \( \sigma \) satisfies \( \phi \), or that \( \sigma \) is a model of \( \phi \). The details of
the structures and the formulae are not relevant for the formalisation.
A formula \( \phi \) is satisfiable over \( \text{Struct} \) if some \( \sigma \) in \( \text{Struct} \) satisfies
\( \phi \), and is unsatisfiable over \( \text{Struct} \) otherwise. We drop the qualifier
“over \( \text{Struct} \)”.

The satisfiability problem is to determine if a given formula \( \phi \) is satisfiable.

We formulate satisfiability in terms of a transformer. The
domain of structures is \( (\mathcal{P}(\text{Struct}), \subseteq, \cup, \cap) \). The model transformer
\( \text{mod}_\phi : \mathcal{P}(\text{Struct}) \to \mathcal{P}(\text{Struct}) \) maps a set of structures to the
subset containing only models of \( \phi \).
\[
\text{mod}_\phi(S) = \{ \sigma \in S \mid \sigma \models \phi \}
\]
Properties of \( \phi \) can be expressed as properties of \( \text{mod}_\phi \). The set
of models of \( \phi \) is \( \text{mod}_\phi(\text{Struct}) \) and \( \phi \) is unsatisfiable exactly if
\( \text{mod}_\phi(\text{Struct}) \) is empty. Observe that \( \text{mod}_\phi \) is completely additive
and reductive, hence it is in the scope of problems we consider.

We denote the function \( (\sim \circ \text{mod}_\phi \circ \sim) \) as \( \text{ucmod}_\phi \) and call
\( \text{ucmod}_\phi \) the conflict transformer. The conflict transformer \( \text{ucmod}_\phi \)
adds to any set of structures all countermodels of \( \phi \). The set of
Theorem 5. A formula $\varphi$ is unsatisfiable exactly if $\text{mod}_{\varphi}(\emptyset)$ contains all structures. Observe that $\text{ucmod}_{\varphi}$ is completely multiplicative and extensive.

**Theorem 6.** Given a transition system, and sets of states $P$ and $Q$, there is no trace from $P$ to $Q$ exactly if $\text{trace}_{P,Q}$ is bottom-everywhere.

A non-bottom witness for $\text{mod}_{\varphi}$ is a set of structures that contains a model of $\varphi$. A minimal non-bottom witness is a singleton set containing a model of $\varphi$. A non-top witness for $\text{ucmod}_{\varphi}$ is a set of structures strictly contained in $\text{Struct}$ that excludes some models of $\varphi$. A maximal non-top witness for $\text{ucmod}_{\varphi}$ is a set that contains all structures except one model of $\varphi$.

We define two logics for use in examples.

**Propositional Logic** Let $\text{Prop}$ be a finite set of propositional variables. The set of literals is $L \doteq \{ p, \neg p \mid p \in \text{Prop} \}$, containing a variable or its negation. A clause is a disjunction of literals and a CNF formula is a conjunction of clauses. As is common in the SAT literature, we view clauses as sets of literals and formulae as sets of clauses. The set of truth values is $B \doteq \{ t, f \}$. Literals, formulae and clauses are interpreted over functions $\text{Asy} \doteq \text{Prop} \to B$, from variables to truth values, and are called propositional assignments.

The entailment relation $\models$ is defined as follows. For each assignment $\sigma$ and literal $l$, $\sigma \models l$ exactly if $\sigma(l) = t$ if $l$ is a variable, and $\sigma(l) = \top$ if $l$ is the negation of a variable. For each clause $C$, $\sigma \models C$ exactly if $\sigma(l) \models C$ for every clause $C$ in $\varphi$.

**Inequality Logic** Inequality logic can express order between variables. Let $\text{Vars}$ be a set of first-order variables. The set of difference predicates is $\{ x < y \mid x, y \in \text{Vars} \}$. The set of inequality literals contains all difference predicates and their negations. We interpret difference formulae over integers. The set of structures $\text{Struct} \doteq \text{Vars} \to \mathbb{Z}$ consists of functions from variables to integers. A structure $\sigma$ satisfies a predicate $x < y$ if the inequality $\sigma(x) < \sigma(y)$ holds. For formulae, the entailment relation $\models$ is defined as expected.

2.3 Feasible Traces via Bottom Everywhere

Let $M = (S, T)$ be a transition system where $S$ is a set of states and $T \subseteq S \times S$ is a transition relation. Let $S^+$ be the set of non-empty sequences of states. A trace is a sequence $\tau = \tau_0, \ldots, \tau_{n-1}$ satisfying that every $(\tau_i, \tau_{i+1})$ is a valid transition. A state $t$ is reachable from $s$ if there exists a trace starting in $s$ and ending in $t$. Given sets of states $P$ and $Q$, the feasible trace problem is to determine if there exists a trace from a state in $P$ to a state in $Q$.

We formulate the feasible trace problem as a bottom-everywhere problem. The domain of sequences is $\mathcal{P}(S^+)$. The feasible trace transformer $\text{trace}_{P,Q} : \mathcal{P}(S^+) \to \mathcal{P}(S^+)$ maps a set of non-empty sequences of states to the subset containing only traces that start from a state in $P$ and reach a state in $Q$. We assume $\tau_n$ is the last state of $\tau$ below.

$$\text{trace}_{P,Q}(X) \doteq \{ \tau \in X \mid \tau \text{ is a trace, } \tau_0 \in P, \tau_n \in Q \}$$

Theorem 7. The following are equivalent, given a transition system and sets of states $P$ and $Q$.

1. No state in $Q$ is reachable from a state in $P$.
2. $\text{trace}_{P,Q}$ is bottom-everywhere.
3. $\text{bread}_{P,Q}$ is bottom-everywhere.
4. $\text{bread}_{P,Q}$ is bottom-everywhere.

The proof of the first three statements is straightforward. The proof of equivalence to 4 is due to Cousot [7]. A minimal non-bottom witness for $\text{trace}_{P,Q}$ is a set of states that is reachable from $P$. A minimal non-bottom witness for $\text{bread}_{P,Q}$ is a state in $P$ from which a state in $Q$ is reachable.

The function $(\neg \circ \text{trace}_{P,Q} \circ \neg)$ maps a set of states $X$ to a set containing $X$ and states that are not reachable from $P$ and states not in $Q$. Observe that such a function can directly be computed by computing a fixed point using the function $\neg \circ \text{post} \circ \neg$. The same applies for $\text{bread}_{P,Q}$ and the function $(\neg \circ \text{pre} \circ \neg)$. These dual functions have been implemented in model checkers and used to compute fixed points. Henzinger et al. [19] discuss temporal properties that can be checked with these fixed points, and Cousot and Cousot [11] combine these fixed points with abstraction.
3. Abstract Procedures for Bottom-Everywhere

In this section, we apply abstraction to the bottom-everywhere problem. If an overapproximation of a function \( f \) on \( \mathcal{P}(S) \) is bottom-everywhere, the function \( f \) is also bottom-everywhere. If an underapproximation of the dual function \( \bar{f} \) is top-everywhere, the function \( \bar{f} \) is also top-everywhere. We now highlight the overapproximate and underapproximate analysis present in SAT solvers and model checkers. We begin by recalling abstract interpretation.

**Abstract Interpretation** The key idea of abstract interpretation is to characterise solutions to a problem by a fixed point and derive approximate solutions by fixed point approximation. For convenience, we work in the Galois connection framework. Cousot and Cousot [10] extensively discuss generalisations. These generalisations are required to analyse theories such as linear arithmetic because the lattice of polyhedra is not complete, and for analysis of automata, and trace-based abstractions.

A Galois connection between posets \((C, \sqsubseteq)\) and \((A, \sqsubseteq)\), written \( A \vdash C \), is a pair of monotone functions \( \alpha : C \to A \) and \( \gamma : A \to C \) satisfying that \( \alpha(x) \sqsubseteq \gamma(\alpha(x)) \) for all \( x \in C \). If \( A \) is an abstract domain and \( C \) is a concrete domain, \( \alpha \) is called the concrete domain and \( C \) is called the abstract domain. Monotone functions on \( C \) are called concrete transformers and those on \( A \) are called abstract transformers. A concrete transformer \( f_a : A \to A \) soundly approximates \( f : C \to C \) if the pointwise order \( f \sqsubseteq \gamma \circ f \circ \alpha \) holds.

If \((C, \sqsubseteq)\) is a powerset lattice, an abstract domain \((A, \sqsubseteq)\) is called an overapproximation because it satisfies \( x \sqsubseteq \gamma(\alpha(x)) \) for all \( x \in C \). An abstract domain \((C, \sqsubseteq)\) is called an underapproximation of \((C, \sqsubseteq)\) because it satisfies \( \alpha(x) \sqsubseteq y \sqsubseteq \gamma(y) \). A sound abstract transformer on an overapproximation is called a sound abstraction transformer.

The next two notions formalise precision of an approximation. A Galois connection implies there is a maximally precise approximation \( \alpha \circ f \circ \gamma \), called the best abstract transformer. An abstract transformer \( f_a \) approximating \( f \) is \( \gamma \)-complete at \( a \) if \( f_a(\alpha(a)) = \gamma(f_a(a)) \), and is \( \alpha \)-complete if it is \( \gamma \)-complete at every \( a \). An abstract transformer \( f_a \) is \( \alpha \)-complete at \( a \), no precision is lost. For example, if \( \alpha \) is a concrete successor transformer and \( \gamma \) is a sound abstraction transformer, then \( \gamma \circ f_a \circ \alpha \) is a sound abstraction transformer.

**Parametric Fixed Points** We require the notion of a fixed point above (or below) an element. Let \( f : L \to L \) be a monotone function on a complete lattice and \( a \) be an element of \( L \). The greatest fixed point below \( a \) denoted \( \text{gfp}(f) \) is the greatest fixed point of the function \( \lambda x. f(x \sqcap a) \). The least fixed point above \( a \) denoted \( \text{lfp}(f) \) is the least fixed point of the function \( \lambda x. f(x \sqcup a) \). The parametric fixed point functions below map an element \( x \in L \) to the least fixed point above \( x \) and greatest fixed point below \( x \), respectively.

\[
\text{plfp}(f) = \lambda x. \text{lfp}_f(x) \quad \text{gfp}(f) = \lambda x. \text{gfp}_f(x)
\]

**Extrapolation and Interpolation** We use the terms extrapolation and interpolation for techniques used to accelerate fixed point computation. Figuratively, extrapolation operators move upwards or downwards from an element in a lattice, as illustrated in Figure 1. Interpolation operators move between elements.

**Figure 1.** Downwards extrapolation to underapproximate a greatest fixed point, and upwards interpolation to underapproximate a least fixed point. The two operations are not dual.

Let \( L \) be a lattice. An upwards extrapolation \( \bigtriangleup : L \to L \) is a unary extensive function. We write \( \bigtriangleup a \) for the application \( \bigtriangleup(a) \). The dual notion of a downwards extrapolation \( \bigtriangledown : L \to L \) is a unary reductive function. The definitions above are based on (but not identical to) those for widening operators without a well-foundedness requirement [10].

A downwards interpolation is a binary function \( \bigtriangledown : L \times L \to L \) satisfying that for all \( x \) and \( y \) in \( L \), the inequality \( x \sqsubseteq y \) implies the inequality \( \bigtriangledown(x, y) \sqsubseteq \bigtriangledown(y, x) \). We write interpolation application as \( x \triangledown y \). The standard definition of a narrowing operator extends downwards interpolation with a well-foundedness condition. The notions of upwards interpolation and dual narrowing, both denoted \( \bigtriangledown \), are dually defined. Note that extrapolation and interpolation are not dual operations, just as widening and narrowing are not dual. Though interpolation is self-dual, we distinguish between upwards and downwards interpolation based on whether the operator is used in a least or greatest fixed point computation.

**Downset Completion** Downset completion is an operation that enriches an abstraction with disjunction [9]. Consider an abstraction \((A, \sqsubseteq, \sqcup, \sqcap)\) of a powerset lattice \( \mathcal{P}(S) \) with concretisation \( \gamma \). The abstraction \( A \) is disjunctive if \( \gamma(a \sqcup b) = \gamma(a) \sqcup \gamma(b) \).

A subset \( Q \) of \( A \) is downwards closed if for every \( x \in Q \) and \( y \in A \), \( y \sqsubseteq x \) implies \( y \in Q \). A downwards-closed set is called a downset. The smallest downset containing \( Q \) is denoted \( Q^- \), and the downset of a singleton set \( \{ x \} \) is denoted \( x^- \). In examples, we denote a downset as the set of its maximal elements. We omit a general discussion of when such a representation is possible. The downset lattice over \( A \), written \( (\mathcal{D}(A), \sqsubseteq, \sqcup, \sqcap) \), is the set of downsets of \( A \) ordered by inclusion. Downsets strictly generalise powersets because the downset lattice with respect to the identity relation is isomorphic to \( \mathcal{P}(S) \).

The downset completion of \( A \) is the lattice \( \mathcal{D}(A) \) with the abstraction and concretisation functions below. In contrast to the standard treatment, we use downsets as underapproximating abstractions.

\[
\gamma_{\mathcal{D}(A)} : \mathcal{D}(A) \to \mathcal{P}(S) \\
\alpha_{\mathcal{D}(A)} : \mathcal{P}(S) \to \mathcal{D}(A)
\]

Consult [9] for proofs that the pairs of functions above form Galois connections and that the domains are disjunctive.

### 3.1 Abstract Bottom-Everywhere

We apply abstract interpretation to determine if a function is bottom-everywhere. Let \((O, \sqcup, \sqcap, \sqsubseteq)\) be an abstract domain in a Galois connection with \((\mathcal{P}(S), \sqsubseteq, \sqcup, \sqcap)\). We denote the abstraction and concretisation functions as \(\alpha_{O}\) and \(\gamma_{O}\). We assume that \(\gamma_{O}(\bot)\) is the empty set.

Let \( f : O \to O \) be a sound abstraction of a completely additive, reductive function \( f \). If \( \text{gfp}(fo) \) concretises to \( \bot \), the function \( f \) is bottom-everywhere. If \( \text{gfp}(fo) \) does not concretise to \( \bot \), we do not know if \( f \) is bottom-everywhere, due to imprecision in the

Note that the functions \( + \) and \( \oplus \) are different. Intuitively, forward-backward iteration exploits a temporal duality between forwards and backwards analysis. In contrast, ACDCCL will exploit a different duality between functions and their De Morgan duals.
We show that the procedure in Algorithm 1 generalises with the first element being either $\perp$ with the first element being either $\perp$ with the first element being either $\perp$ with the first element being either $\perp$. If we know that $f_o$ is $\gamma$-complete at some element $a$, we can determine, despite working in an abstraction, that $f$ is not bottom-everywhere. Since $f_o$ only has to be $\gamma$-complete at a single element, the lattice and transformer may still be imprecise.

**Proposition 9.** If $f_o$ is $\gamma$-complete at $a$ and $\gamma(f_o(a))$ is not $\perp$, $f$ is not bottom-everywhere.

The procedure `Abstract-non-⊥` below takes as input an overapproximate transformer $f_o$, an abstract element $o$ and a downwards extrapolation operator $\triangledown$. In addition, a procedure to check emptiness $\gamma(o) = \emptyset$ is required together with a sufficient criterion for $\gamma$-completeness of $f_o$. The procedure attempts to determine if $f_o$ is bottom on all elements below $o$ in the lattice. The output is a pair with the first element being either $\perp$, not $\perp$ or unknown, and the second element representing the last lattice element obtained.

If $\text{pgf}(f_o)(o)$ concretises to $\perp$, the function $f$ is bottom on elements below $\gamma(o)(o)$. If $f_o$ is not bottom on the fixed point $o'$ and is $\gamma$-complete at $o'$, we know $f$ is not bottom-everywhere. Neither condition above may hold due to imprecision in the transformer or domain. In this case, downwards extrapolation is used to check if $f_o$ is also bottom on elements below the fixed point. Unlike standard applications of widening, downwards extrapolation is applied here to improve precision.

**Algorithm 1:** `Abstract Search for a Non-⊥ Witness`

\[
\text{Abstract-non-⊥}(f_o : O \rightarrow O, \triangledown : O \rightarrow O, o : O) \rightarrow O
\]

\[
\text{repeat} \quad | \quad o' \leftarrow o \\
\text{until} \quad o = o' \cap f_o(o) = \emptyset \\
\text{if} \quad \gamma(o) = \emptyset \text{ then return } (\perp, o') \\
\text{if} \quad f_o \text{ satisfies } \gamma \text{-completeness criterion at } o \text{ then} \\
\quad \text{return} \quad \text{not } (\perp, o) \\
\quad d \leftarrow \triangledown o \\
\quad \text{if} \quad d = o \text{ then return } \text{unknown}, d \\
\text{return} \quad \text{Abstract-non-⊥}(f_o, \triangledown, d)
\]

**Model Search in SAT Solvers**

We show that the procedure in Algorithm 1 generalises model search in SAT solvers. The abstract lattice contains partial assignments, the abstract transformer is called the unit rule, the greatest fixed point computation $\text{pgf}(f_o)$ is Boolean Constraint Propagation (BCP) and downwards extrapolation is implemented by decisions. We briefly elaborate on these points. Further details are in [13].

A partial assignment maps each variable to true, false, or unknown. The set of partial assignments $\text{PAsg} \overset{\triangle}{=} \{\text{Vars} \rightarrow \{\text{true, false, } \top\}\}$ consists of partial assignments and a unique element $\perp$. In implementations of SAT solvers, the state $\perp$ is indicated by a special conflict flag. Partial assignments form a lattice with respect to the natural pointwise order and are equivalent to the constants lattice or the Cartesian abstraction for Boolean valued variables. The element $\top$ represents a partial assignment in which all variables have undefined values, and $\perp$ represents the empty set. The concretisation function $\gamma : \text{PAsg} \rightarrow \mathcal{P}(\text{Asg})$ maps $\perp$ to $\emptyset$, and $\pi \neq \perp$ to the set defined below.

\[
\gamma(\pi) \triangleq \{\sigma \mid \text{for all } x \in \text{Vars}, \pi(x) \neq \top \text{ implies } \pi(x) = \sigma(x)\}
\]

Propositional solvers deduce properties about a formula using the unit rule. The unit rule asserts that if a partial assignment is defined on all but one literals in a clause and does not satisfy those literals, it must satisfy the remaining literal to satisfy the formula. For a propositional literal $l$, we write $\pi \models l$ if $l \models x$ and $\pi(x) = t$ or if $l = \neg x$ and $\pi(x) = f$. Lattice-theoretically, the unit rule for a clause is a decreasing transformer on the lattice of partial assignments.

**Proposition 9.** If $f_o$ is $\gamma$-complete at $a$ and $\gamma(f_o(a))$ is not $\perp$, $f$ is not bottom-everywhere.

The concretisation function $\gamma : \text{PAsg} \rightarrow \mathcal{P}(\text{Asg})$ maps $\perp$ to $\emptyset$. The empty set represents the empty set. We briefly express.

\[
\text{Unit}_{C}(\pi) \overset{\triangle}{=} \left\{ \begin{array}{ll}
\perp & \text{for all } l \in C, \pi(l) = \neg t \\
\pi \cap \{x \rightarrow t\} & \text{for all } l \in C', \pi(l) = t \\
\pi \cap \{x \rightarrow f\} & \text{for all } l \in C', \pi(l) = \neg t \\
\pi & \text{otherwise}
\end{array} \right.
\]

Moreover, $\text{Unit}_{C}$ can be characterised as the best abstract transformer for the model transformer $\text{modc}$. The unit rule extends to formulae in CNF by taking the pointwise meet of unit rules for each clause in the formula.

\[
\text{Unit}_{\varphi} \overset{\triangle}{=} \bigcap_{C \in \varphi} \text{Unit}_{C}
\]

Boolean Constraint Propagation (BCP) repeatedly applies the unit rule and computes the fixed point $\text{pgf}(\text{Unit}_{\varphi})$. A solver makes a decision by assuming that some variable, which is unknown in a partial assignment, has a definite value. Mathematically, a decision maps a partial assignment $\pi$ to $\pi[x \rightarrow v]$, where $x$ is unknown in $\pi$ and $v$ is a truth value. Observe that decisions are applications of a downward extrapolation operator $\triangledown : \text{PAsg} \rightarrow \text{PAsg}$. If all variables are assigned to $t$ or $f$ in $\pi$, then $\triangledown_{\perp} \pi = \pi$. Otherwise, $\triangledown_{\perp} \pi$ is $\pi[x \rightarrow t]$ or $\pi[x \rightarrow f]$ for some $x$ such that $\pi(x) = t$.

Consider the conditions in Abstract-non-⊥. If BCP starts from $\top$ and leads to $\perp$, the formula is unsatisfiable, as returned by the first conditional. A SAT solver terminates if it finds a satisfying assignment. Since assignments are partial assignments, the unit rule applies to them. Observe that $\text{Unit}_{\varphi}(\pi)$ for an assignment is $\perp$ if $\pi$ does not satisfy $\varphi$ and is $\pi$ otherwise. More generally, $\gamma(\text{Unit}_{\varphi}(\pi))$ equals $\text{modc}(\gamma(\pi))$, so finding a satisfying assignment is a sufficient condition for $\gamma$-completeness. Another sufficient condition is that at least one literal in each clause is satisfied by the current partial assignment.

To summarise, the algorithmic content of model search in a SAT solver is an instance of Abstract-non-⊥, where $O = \text{PAsg}$, $\triangledown$ is propositional decision making, $f_o$ is given by $\text{Unit}_{\varphi}$ and the $\gamma$-completeness check tests whether all variables have been assigned to $t$ or $f$. Several heuristics and carefully implemented data structures are required to achieve high performance with BCP, but these aspects correspond to optimising the fixed point iteration and abstract domain implementation.

**Model Search for Inequalities**

We instantiate Abstract-non-⊥ for reasoning about inequality formulae. Let $\text{Lit}$ be the set of inequality literals. We introduce an inequality abstract domain, which contains only conjunctions of inequalities. Specifically, each element of $(\text{Ineq}, \triangledown, \cup, \cap)$, where $\text{Ineq} = \mathcal{P}(\text{Lit})$ represents a conjunction of inequalities. We use the superset order because a larger set of constraints represents more constraints and has fewer models. The join is intersection of sets of constraints and meet is union. The empty set represents true and the set of all inequalities is one representation of false. Observe that false has multiple representations because every set containing a predicate and its negation is equivalent to false.
Recall that the concrete domain of structures was $P(Vars \rightarrow \mathbb{Z})$. The concretisation function $\gamma_{\text{Ineq}} : \text{Ineq} \rightarrow P(Vars \rightarrow \mathbb{Z})$ maps a set of constraints to their models.

$$\gamma_{\text{Ineq}}(\pi) = \{\sigma : \sigma \models P, \text{for every } P \text{ in } \pi\}$$

We use an example due to McMillan et al. [23] to illustrate Abstract-non-$\bot$ on an inequality formula.

$$\varphi \doteq (a < b) \land (a < c) \land (b < d \lor c < d) \land (d < a)$$

As with propositional logic, we will use the best abstract transformer for a clause, but construct the transformer for a formula in the abstract.

$$\text{amod}_\varphi \doteq \bigwedge_{C \in \varphi} \alpha_{\text{Ineq}} \circ \text{mod}_C \circ \gamma_{\text{Ineq}}$$

We apply Abstract-non-$\bot$ to check satisfiability of $\varphi$. The sets below represent the evaluation of each clause in the abstract domain.

$$\text{amod}_\varphi(\emptyset) = \{a < b\} \cup \{a < c\} \cup \emptyset \cup \{d < a\}$$

$$\text{amod}_\varphi(\pi) = \{a < b, a < c, d < a\}$$

\text{amod}_\varphi(\pi) \cup \{d < b, d < c\} \cup \{\text{amod}_{b<d}(\pi) \cap \text{amod}_{b<d}(\pi)\} = \text{Lit}$$

The second step above requires explanation. The best abstract transformer for the singleton clauses when applied to $\pi$ has the effect of computing the transitive closure of constraints in $\pi$. Due to $(a < b)$ and $(d < a)$, the inequality $(d < b)$ is added to $\pi$. The conjunction of $\pi$ with $b < d$, represented by $\text{amod}_{b<d}(\pi)$, is unsatisfiable, as is the conjunction of $\pi$ with $c < d$. The best representation of an unsatisfiable conjunction is the set of all constraints.

Compare the calculation above to solving the same formula with DPLL($T$). In DPLL($T$), a propositional variable would have been introduced for each predicate, and two solvers, one for propositional logic and one for inequalities would have been required to solve the formula. By instantiating Abstract-non-$\bot$, we can solve the formula using only the abstract domain, which plays the role of a theory solver, and without introducing extra variables. This difference becomes important for large formulae and is the motivation for techniques like natural domain SMT [5].

The generalised DPLL technique of McMillan et al. [23], referred to as GDPLL, also solves this formula without introducing propositional variables. Unlike ACDDL, GDPLL makes decisions and uses the shadow rule for conflict analysis to solve the formula.

An instantiation of Abstract-non-$\bot$ over Ineq may use amod as defined above for fo, a decision operator $\triangledown_? : \text{Ineq} \rightarrow \text{Ineq}$ such that $\triangledown_? \text{Ineq} = \text{Ineq}$ and otherwise $\triangledown_? \pi = \pi \cup \{x < y\}$ such that $x < y$ is not in $\pi$. The $\gamma$-completeness check for an element $\pi$ can be performed by determining whether for every clause $C \in \varphi$, the intersection $\pi \cap C$ is non-empty. The check for $\gamma(a)$ being empty amounts to detecting a conjunction of unsatisfiable constraints. If constraints are represented as directed graphs, the emptiness check is implemented by cycle detection.

**Feasible Traces** The automata-theoretic approach to the feasible traces problem is to construct an automaton representing the traces of a system and the negation of a correctness property, and check if the language of this automaton is empty. This approach can be viewed as indirectly computing the greatest fixed point of $\text{trace}_{P,Q}$, where $P$ represents the set of initial states of a system, and $Q$ represents states that violate a property.

### 3.2 Abstract Top-Everywhere

CDCL is distinguished from several procedures that combine dual reasoning because the procedure used to reason about the top-everywhere problem is not dual to Abstract-non-$\bot$. Let $(U, \geq, \lor, \land)$ be an abstract domain that underapproximates $(P(Vars), \subseteq, \cup, \cap)$ with functions $\alpha_{\varphi}$ and $\gamma_\varphi$. We assume that $\gamma_\varphi(T)$ is $S$. Let $fu : U \rightarrow U$ be a sound abstraction of a completely multiplicative, extensive function $f$, where $f$ is completely additive and reductive. If $\text{lp}(fu)$ concretises to $T$, then the function $f$ is bottom-everywhere.

The procedure Abstract-non-$\top$ computes an underapproximation of a least fixed point using the upwards interpolation operator $\triangledown_\varphi$. Correctness follows directly from known theorems in abstract interpretation and it suffices for us to discuss instantiations of this procedure.

### Algorithm 2: Abstract non-$\top$

**Algorithm 2:** Abstract non-$\top$

**Abstract-non-$\top$**($fu : U \rightarrow U, \Delta t : U \times U \rightarrow U, u : U$)

$c \leftarrow u \Delta t(u \cup fu(u))$

If $\gamma(c) = \top$ then return $(\top, c)$

If $fu$ satisfies $\gamma$-completeness criterion at $c$ then

return (not $\top, c$)

If $c = u$ then return (unknown, $c$)

return Abstract-non-$\top$(fu, $\Delta t$, c)

### Conflict Analysis in SAT Solvers

We now show that conflict analysis in SAT solvers is an instance of Abstract-non-$\top$. Consider the downset completion $(\triangledown(PAsg), \subseteq)$ of partial assignments treated as an underapproximation. That is, every set of truth assignments is underapproximated by a set of partial assignments. Conflict minimisation [26] is a technique used by SAT solvers to generalise the reason behind a partial assignment leading to a conflict. Minimisation techniques replace a partial assignment $\pi$ with partial assignments from which $\pi$ can be derived by the unit rule. There may not be a unique partial assignment from which $\pi$ is derived, so downsets of partial assignments have to be considered. We define a transformer for conflict minimisation below.

$$\text{minimise}_\varphi : \triangledown(PAsg) \rightarrow \triangledown(PAsg)$$

$$\text{minimise}_\varphi(\pi) = \pi \cup \{x \mid \text{Unit}(c)(\pi) \in Q \text{ for some } C \in \varphi\}$$

Conflict minimisation can dually be viewed as restricted application of resolution.

**Example 1.** Consider a formula $\varphi \doteq \theta \land (x \lor \neg y) \land (\neg x \land y)$ and a partial assignment $\pi = \{x \mapsto \text{true}, y \mapsto \text{true}\}$ such that $\text{pgp}(\text{Unit}(\pi)) = \bot$. Since $y$ can be derived by the unit rule if $x$ is true and $x$ can be derived if $y$ is true, we have $\text{minimise}_\varphi(\pi) = \{\{x \mapsto \text{true}\}, \{y \mapsto \text{true}\}\}$.

There may be many ways to minimise a conflict but generating all minimisations exhausts solver memory. To avoid representing sets of conflicts, a SAT solver usually chooses one conflict. In Example 1, each set in $\Pi = \{\{x \mapsto \text{true}\}, \{y \mapsto \text{true}\}\}$ can be used. We model this step with a choice function $\text{choice} : \triangledown(PAsg) \times \triangledown(PAsg) \rightarrow \triangledown(PAsg)$. The choice function ensures that the effort of generalising a conflict is not lost, and satisfies $\Pi \subseteq \text{choice}(\Pi, \Pi') \subseteq \Pi'$, where $\Pi$ and $\Pi'$ are downsets. Observe that choice is an upwards interpolation.

We emphasise that the soundness of the procedures in this section follows from standard results in abstract interpretation.

Over programs, instances of Abstract-non-$\top$ are procedures that underapproximate least fixed point computations. For example, the set of all counterexamples leading to an error can be defined by a least fixed point. Counterexample analysis techniques usually underapproximate this set by heuristically choosing one, or some subset of counterexamples.
4. Abstract Conflict Driven Learning

The procedures in the previous section were treated separately. The

CDCL algorithm in modern solvers combines Abstract-non-⊥ and
Abstract-non-⊤ as summarised in Figure 2. Rather than return
information from Abstract-non-⊥, information from the fixed point
computation drives Abstract-non-⊤. If Abstract-non-⊤ produces
inconclusive results, Abstract-non-⊥ can still “learn” information
about the conflict for the greatest fixed point computation. This
section makes this combination precise.

The Abstract Conflict Driven Learning procedure (ACDL) is
shown in Algorithm 3. The procedures alternates runs of Abstract-
non-⊥ and Abstract-non-⊤. Communication between the two pro-
cedures is achieved using two functions, fou and learn. Conflicting
elements are transferred from Abstract-non-⊥ to Abstract-non-⊤
using a function fou. We require that this function soundly underap-
proximates f, i.e., that \( f \circ \gamma \circ O \subseteq \gamma \circ \gamma \circ O \). A natural choice for this
transformer is to compute the composition \( \alpha \circ \gamma \circ O \), which maps an
abstract element in \( O \) to its best underapproximation in \( U \). In the
other direction, a transformer is learnt, as discussed below.

Algorithm 3: Abstract Conflict Driven Learning

\[
\text{ACDL}(f_o : O \rightarrow O, \forall \gamma : O \rightarrow O, \\
f_u : U \rightarrow U, \Delta_u : U \times U \rightarrow U, \\
fou : O \rightarrow U, \text{learn} : U \times U \rightarrow (O \rightarrow O)) \\
\begin{algorithmic}
\State \textbf{loop}
\State \( (s, o) \leftarrow \text{Abstract-non-⊥}(f_o, \top, \forall \gamma) \)
\State \textbf{if} \( s \neq \bot \) \textbf{then return} \( (s, o) \)
\State \( u \leftarrow \text{fou}(o) \)
\State \( (s, u) \leftarrow \text{Abstract-non-⊤}(f_u, u, \Delta_u) \)
\State \textbf{if} \( s = \top \) \textbf{then return} \( (\bot, u) \)
\State \( fo \leftarrow fo \cap \text{learn}(u) \)
\end{algorithmic}
\]

Best Learning Transformer If Abstract-non-⊥ derives \( \bot \) start-
ing from \( o \), to determine if \( f \) is bottom at a concrete element \( c \), it
suffices to check if \( f \) is bottom at \( c \triangleleft \gamma \circ O(o) \), because \( f \) is reductive.
If Abstract-non-⊥ is invoked on an abstract element \( o \), it suffices
to check if \( f \) is bottom on elements below \( \alpha \circ \gamma \circ o (\gamma \circ O(o) \cap \gamma \circ v(u)) \).
The procedure Abstract-non-⊤ is used to generalise an element that
leads to a conflict but it operates in an underapproximating domain.
Thus, we need to learn information about overapproximate elements
after generalising within an underapproximation.

The best learning transformer is a function that takes elements \( u \)
and \( o \) and subtracts \( u \) from \( o \).

\( \text{learn} : U \times O \rightarrow O \quad \text{learn}_u(o) \triangleq \alpha \circ (\gamma \circ o (\gamma \circ O(o) \cap \gamma \circ v(u))) \)

Theorem 10. Let \( f_o \) be a sound overapproximation of a completely
additive, reductive transformer \( f \) and let \( f_u \) be a sound underap-
proximation of the De Morgan dual \( f \) of \( f \). If \( p \in p \in f(o) \) \( \alpha \circ (\gamma \circ O(o)) \)
is a sound overapproximation of \( f \).

Theorem 10 strictly generalises the notion of learning from SAT
solvers to every instance of the bottom-everywhere problem.

We give an example of a simple, sound learning procedure
supported by all lattices. An element that leads to \( \bot \) is \( tabu \), in
the sense of tabu search. Tabu learning defines a sound learning
transformer \( Tabu : U \times O \rightarrow O \).

\[
Tabu\circ o(a) = \begin{cases} 
\bot & \text{if } \gamma(o) \subseteq \gamma(u) \\
o & \text{otherwise}
\end{cases}
\]
The tabu rule checks whether search has entered a region that is
known to map to \( \bot \).

Theorem 11 (Soundness). If ACDL returns not \( \bot \), then \( f \) is not
bottom everywhere. If it returns \( \bot \), then \( f \) is bottom everywhere.

Proof. Assume the algorithm returns not \( \bot \). Then Abstract-non-⊥
returns (not \( \bot \), \( o \)). Then \( f_o \) is \( \gamma \)-complete at \( o \), \( \gamma \circ o \neq \bot \) and
\( f_o(o) = o \). By \( \gamma \)-completeness, we then conclude that \( f \) is not
bottom at \( \gamma(o) \).

Assume the algorithm returns \( \bot \). Then Abstract-non-⊤ re-
turned \( \top \). We denote by \( u \) the initial value of \( u \) in Abstract-
non-⊤, and by \( u' \) the return value with \( \gamma\circ u' = \top \). It is an
invariant of the algorithm that \( f \) is a sound overapproximation of \( f \). Hence, whenever Abstract-non-⊥ returns (\( \bot \), \( o \)) it holds that
\( f(\gamma(o)) = \bot \). It holds that \( u = \alpha \circ \gamma \circ O(o) \) for such an \( o \). By
duality, we have that \( f(\bot) \supseteq \gamma(\gamma(o)) \supseteq \gamma(u) \). Abstract-non-⊤
returns \( u' \supseteq u \) such that \( \gamma(u') = \top \). By soundness of abstract
interpretation it holds that \( \gamma(u') = f(\gamma(u)) = \top \). Since \( f \) is an
upper closure operator with \( f(\bot) \supseteq \gamma(u) \) and \( f(\gamma(u)) = \top \),
it follows by idempotence that \( f(\bot) = \top \). Dually, it follows that
\( f(\top) = \bot \). Therefore, \( f \) is bottom everywhere.

5. Abstract Conflict Driven Clause Learning

The previous section showed that the notion of learning is very
general. Clause learning is a specific form of learning in which
abstract transformers are implicitly represented by clauses. Since
model search in SAT solvers is driven by the unit rule, clause learn-
ing can be viewed as learning unit rule transformers. We present
a generalised unit rule that lifts clause learning to richer lattices.
Abstract Conflict Driven Clause Learning (ACDCL) is a strict gener-
alisation of the CDCL algorithm in SAT solvers. In § 3, we demon-
strate that propositional solvers operates over the overapprox-
imate partial assignments domain \( PAsg \) and its underapproximate
downset completion \( \mathcal{D}(PAsg) \). We present ACDCL as a variant of the
ACDL procedure presented in the previous section operating over an
overapproximation \( O \) and its downset completion \( \mathcal{D}(O) \).

5.1 Generalising Clause Learning

To understand the lattice-theoretic essence of clause learning, it is
useful to compare the unit rule with tabu learning. Consider a clause
\( C = p \lor \neg q \) and the partial assignment \( \pi = \{ p \rightarrow f, q \rightarrow t \} \),
which contains no satisfying assignment to \( C \). Consider \( \pi ' \) strictly
greater than \( \pi \). We have that \( \text{Tabu}_u(\pi ') = \pi ' \). However, if \( \pi ' \) is the
partial assignment \( \{ p \rightarrow \text{false} \} \), by unit rule application we have
\( \text{Unit}(\pi ') = \{ p \rightarrow \text{false}, q \rightarrow \text{false} \} \). The tabu rule only drives
search away from elements where \( f \) is bottom. In contrast, if \( f \) is
An abstraction $A$ with complementable meet decompositions admits a generalised unit rule $\text{gunit}_A : A \times A \rightarrow A$. In contrast to the propositional unit rule, which is defined with respect to a clause, we define the generalisation with respect to elements that lead to $\bot$. This is only a difference of presentation, since the negation of a clause is a partial assignment on which $\text{Unit}_\bot$ is bottom.

$$\text{gunit}_A(a) = \begin{cases} \bot & \text{for all } m \in \text{mdc}(c), a \subseteq m \\ a \cap \bar{\Pi} & \text{mdc}(c) = M \cup \{n\} \\ a & \text{and for all } m \in M, a \subseteq m \end{cases}$$

If $\text{gunit}_A(a)$ returns $a \cap \bar{\Pi}$ as above, we call $\Pi$ a unit irreducible.

The transformer $\text{gunit}$ can be lifted to a learning transformer $\text{gunit} : O \times U \rightarrow O$, where $U = \mathcal{Q}(O)$. For a downwards-closed set $S$, we define $\text{gunit}_S = \bigcap_{\alpha \in S} \text{gunit}_\alpha$. We skip the proof that $\text{gunit}$ is a sound learning transformer. We instantiate abstract conflict driven learning with $\text{gunit}$ to derive Abstract Conflict Driven Clause Learning.

### 5.2 An Abstract Backjumping Algorithm

Abstract conflict driven learning is sound. To achieve completeness, we need to ensure that learning drives the Abstract-non-$\bot$ procedure to explore new regions of the abstract lattice. The abstract backjumping algorithm appearing next generalises the backjumping search of propositional CDCL. The abstract procedure in Algorithm 4 operates over an overapproximate domain $O$ with complementable decompositions, and its underapproximate downset completion $\mathcal{Q}(O)$. Learning refines the transformer $fo$ with the generalised unit rule $\text{gunit}$.

#### Algorithm 4: Backjumping and Clause Learning

```
ACDCL(fo : O \rightarrow O, \\
\n\mathcal{N} : O \rightarrow O, \\
fo : \mathcal{Q}(O) \rightarrow \mathcal{Q}(O), \\
\n\Delta : \mathcal{Q}(O) \rightarrow \mathcal{Q}(O))
```

Initialize $\rho \in O^*$ to singleton sequence $\top_O$

repeat

\( (s, \rho) \leftarrow \text{Abstract-non-}\bot^*(fo, \rho, \mathcal{N}) \)

if $s \neq \bot$ then return $(s, \prod \rho)$

$u \leftarrow \{\prod \rho\}$

$(s, (c)) \leftarrow \text{Abstract-non-}\neg(fo, u, \Delta t)$

$l \leftarrow \text{gunit}_A$

$\rho \leftarrow \text{backjump}(l, fo, \rho)$

$fo \leftarrow \text{not} l$

until $\rho$ is empty

return not $\bot$

```
Abstract-non-\bot^*(fo : O \rightarrow O, \mathcal{N} : O \rightarrow O, \rho : O^*)
```

\( o \leftarrow \prod \rho \)

repeat

\( o' \leftarrow o; o \leftarrow fo(o) \cap o \)

until $o = o'$ or $\gamma(o) = \emptyset$

if $\gamma(o) = 0$ then return $(\bot, \rho)$

$\rho \leftarrow \rho \cdot o$

if $fo$ satisfies $\gamma$-completeness criterion at $o$ then

return (not $\bot, \rho$)

\( d \leftarrow \nabla l_o \)

if $d = o$ then return (unknown $o$)

return Abstract-non-\bot^*(fo, \rho \cdot d, \mathcal{N})
```

The algorithm uses a modification of Abstract-non-$\bot$ to record transformer application on a stack $\rho$. Concatenation of two stacks
\(\rho\) and \(\rho'\) is denoted by \(\rho \cdot \rho'\). The difference to Abstract-nonn⊥ is that after learning backjumping resets the procedure to a state before the element that is negated by the learning transformer. A call to backjump\((l, fo, \rho)\) returns a non-empty prefix \(\rho'\) of \(\rho\) such that \(\gamma(fo'((\rho'))) \neq \bot\) where \(fo' = l \cap fo\). If no such prefix exists, the empty sequence is returned, indicating that \(fo\) is bottom-everywhere.

**Progress and Precision** If we disregard extensions such as restarts and forgets, propositional CDCL never generates the same clause twice. The number of clauses over a fixed set of literals is finite, so a CDCL solver makes constant progress and necessarily terminates. The procedure ACDCL makes progress if each iteration of the outer loop decreases some well-founded order. CDCL solvers do not return unknown. Decisions refine partial assignments until either \(\bot\) or a satisfying assignment is obtained.

In propositional solvers, decisions are made by computing a greatest lower bound between the current element and a meet irreducible, that is, a partial assignment in which only one variable is \(\top\). Decisions never directly cause a conflict. We generalise these conditions below. A meet irreducible extrapolation is a downwards extrapolation operator \(\square\) : \(O \rightarrow O\) satisfying two conditions.

1. \(\square o\) equals \(o\) for some meet-irreducible \(m\).
2. If \(\gamma(\omega) = \bot\), then \(\gamma(\omega) = \bot\).

The precision requirement given next ensures that a sequence of decisions eventually leads to an abstract element that can be analysed without loss of precision. A meet-irreducible extrapolation is precise if whenever \(o\) is equal to \(\square o\), the function \(fo\) is \(\gamma\)-complete at \(o\) with respect to \(f\).

The progress condition in a propositional solver is that learning causes model search to deduce new information. The generalised notion we use is that of asserting backjumps, which ensures that the procedure navigates away from the element that most recently led to \(\bot\). A backjump function is asserting if whenever backjump\((l, fo, \rho)\) returns \(\rho'\), then \(\rho'\) is the empty sequence or \(l(\rho')\) is strictly smaller than \(\rho\). This condition ensures that backjumping drives the search into a new part of the search space.

**Lemma 12.** In some non-final iteration of the ACDCL main loop, let \(\rho\) and \(\rho'\) be, respectively, the stack before and after backjumping. If the backjump function is asserting, then after learning, it holds that \(fo(\rho') \subseteq fo(\rho)\) and \(\rho\) and \(\rho'\) are incomparable.

**Proof.** Let \(o\) and \(c\) be, respectively, \(\rho\) and \(\rho'\), and let \(\{e\}\) be the element returned by Abstract-nonn⊥. It holds that \(c \sqsubseteq o\). Let \(gunit_\omega(o) = o' \cap m\) where \(m\) is the unit irreducible and therefore after learning \(fo(o') \subseteq m\). It holds that \(m \sqsupseteq c\) and therefore also \(m \sqsupseteq o\).

We now show that \(fo(o')\) and \(o\) are not ordered. Assume for a contradiction that \(fo(o') \subseteq o\). Then it holds that \(fo(o') \subseteq m\). Since we also know that \(fo(o') \subseteq m\) it must hold that \(\gamma(fo(o')) = \emptyset\).

This violates the condition that the backjumping function returns a non-conflicting prefix \(\rho'\). Therefore \(fo(o') \not\subseteq o\). Now assume for a contradiction that \(o \sqsubseteq fo(o')\). Then \(o \sqsubseteq m\), and from \(o \sqsubseteq c\) we can derive \(o \sqsubseteq m\) and consequently \(\gamma(o) = \bot\). This is not possible because Abstract-nonn⊥ never returns a trail representing the empty set. We have shown that \(o\) and \(o'\) are incomparable.

Analogous to the case of propositional SAT, asserting backjumps can be implemented using a meet-irreducible extrapolation operator. We informally sketch the reason: Let \(\rho\) be the stack \(\rho_1 \ldots \rho_k (\rho_k \cap m)\) during a run of the procedure such that the last element \((\rho_k \cap m)\) is derived by applying a meet-irreducible extrapolation operator. If we can determine that \(f(\gamma(\rho_k)) = \emptyset\), we may refine \(fo\) by learning the transformer \(\operatorname{gunit}(\rho_k)\). Backjumping to \(\rho_1 \ldots \rho_k\) and then applying \(\operatorname{gunit}(\rho_k)\) yields the element \(\rho_k \cap m\), which drives the search to a new region.

**Theorem 13 (Relative Completeness).** If the downwards extrapolation \(\square\) is precise, and ACDCL\((fo, \square, fu, \Delta)\) terminates, the result is \(\bot\) or not \(\bot\).

**Proof.** We prove by contradiction. If unknown is returned, Abstract-nonn⊥ returned (unknown, \(o\)), which is only possible if \(\square o\) equals \(o\). Since \(\square\) is precise, we know that \(fo\) is \(\gamma\)-complete at \(o\). This is impossible because not \(\bot\) would have been returned.

**Theorem 14 (Termination).** If \(O\) is finite and backjumps are asserting, then ACDCL makes progress.

**Proof.** The procedures Abstract-nonn⊥ and Abstract-nonn\(\top\) terminate over finite lattices. To prove the theorem, we assume that the outer loop of ACDCL is non-terminating and derive a contradiction.

Let \(r\) be the last element of the sequence \(\rho\) at the beginning of an arbitrary iteration of the main loop of ACDCL. Since the lattice is finite, we can reason using well-founded induction. We show that backjumping eventually resets the stack to a prefix whose last element is \(b \sqsubseteq r\).

**Base step:** Assume \(r\) is an atom, meaning every \(r'\) that satisfies \(r' \sqsubset r\) also satisfies \(r' = \bot\). Then the Abstract-nonn\(\top\) step will be initialised with \(\{r\}\), and will return an element \(c \sqsubseteq r\). Therefore, it holds that after learning, \(fo(r) = \bot\). Backjumping therefore necessarily returns to an element \(b \sqsubseteq r\) in order to satisfy the condition that after a backjump \(fo(b) \not\subseteq r\).

**Induction step:** Assume that for all \(r' \sqsubset r\), the induction hypothesis holds. Assume for a contradiction that backjumping never jumps to an element greater than \(r\). Then \(r\) is the target of a backjump infinitely often. Then it must be the case that in two subsequent iterations \(i_1\) and \(i_2\), Abstract-nonn⊥ returns a stack with the same final conflicting element \(c\) such that \(r\) is the target of the corresponding backjump in both cases. After the first such
backjump, we have by Lemma 12 that $f_o$ immediately infers an element $a \sqsubseteq r$ such that $a$ and $c$ are unordered. Specifically, $a$ is not greater than $c$. Therefore, any subsequent call to Abstract-non-\textsubscript{\textcircled{.}}$^+$ returns a stack with a last element that is smaller than $a$, and therefore different from $c$. This contradicts the above, which states that $i_2$ backtracks from $c$.

This completes the proof. It therefore holds that, in an infinite run, backtracking will eventually return an element strictly greater than $\top$. This is impossible, hence all runs of AC\textsubscript{DCL} terminate.  

Though we only show termination for finite lattices, the termination of CD\textsubscript{CL} is usually non-trivial to argue. When applied to decidable logics that involve infinite lattices (such as linear arithmetic), specific details of the theory, the transformers and acceleration techniques must be used to prove termination. It is not clear that there are general termination arguments for infinite lattices. Ascending and descending chain conditions are not enough because of the alternation of the two procedures and the use of decisions and choice, which also operate on infinite sets.

6. Abstract Implication Graphs

The design of underapproximate transformers, such as $\tilde{f}_u$, is a challenging problem that has received less attention than the design of overapproximate transformers. SAT solvers use $f_o$ applications to construct an implication graph, and use this graph to derive $f_u$. We now generalise implication graphs to other domains.

Implication Graphs as Transformer Abstractions Consider a proof rule that when applied to antecedents $a_1, \ldots, a_n$ yields the consequent $c$. If we view antecedents and consequents as elements of an abstract domain, we can formalise a proof rule as a transformer $f_o$ that satisfies $f_o(\prod\{a_1, \ldots, a_n\}) \sqsubseteq c$. Implication graphs provide a means to derive antecedents from consequents thereby implementing an abduction transformer.

Recall that $M_O$ is the set of meet irreducibles of $O$. An implication edge is a directed hyperedge in $E_I \equiv \powerset(M_O) \times M_O$. We order implication edges so that $E_1 \succeq E_2$ means $E_1$ requires weaker antecedents than $E_2$ to derive stronger consequents. Define $(M_1, m_1) \leq (M_2, m_2)$ to hold if $m_1 \sqsubseteq m_2$ and for every $m_1'$ in $M_1$, there exists $m_2'$ in $M_2$ satisfying $m_1' \sqsubseteq m_2'$. An implication graph $G$ is an upwards-closed set of implication edges and the domain of implication graphs $\mathcal{I} \equiv \{\gamma(E_I), \sqsubseteq, \cup, \cap\}$ contains implication graphs with the superset order. In practice, an upwards-closed set is represented by its minimal elements.

Figure 5 contains examples of implication graphs. All incoming edges to a node represent one hyperedge and each hyperedge represents its upwards-closure. A hyperedge of the form $(\{\}, m)$ is depicted by $m$.

An implication graph represents an abstraction of a transformer, as shown below. Assume below that $f_o$ is a reductive transformer. Let $(\text{Red}_O, \sqsubseteq)$ be the lattice of reductors on $O$ with the pointwise order. Define two functions $\alpha \gamma : \text{Red}_O \rightarrow \mathcal{I}$ and $\gamma \mathcal{I} : \mathcal{I} \rightarrow \text{Red}_O$ below.

$$\alpha \gamma (f) \equiv (\{M, m\} \in E_I \mid f(\prod M) \sqsubseteq m)$$

$$\gamma \mathcal{I} (I) \equiv \lambda \alpha \gamma \prod\{m \mid \exists (M, m) \in I. \prod M \sqsubseteq o\}$$

Proposition 15. The functions $\alpha \gamma$ and $\gamma \mathcal{I}$ form a Galois connection.

Example 3. We illustrate the application graph construction for reachability analysis of a program with the interval abstract domain. The problem is to determine if the location $i$ is reachable in the flow graph in Figure 6. An implication graph derived by applying abstract successor transformers is shown alongside.

The label DL\textsubscript{0}, denotes decision level 0, which contains meet irreducibles that are deduced without making assumptions. These represent facts obtained by a run of a static analyser. Due to imprecision, the error is reachable in the abstract.

Suppose we apply downwards extrapolation to force the constraint $a \leq -42$ at the location $n_1$. This constraint is shaded in the figure. If we continue static analysis with this assumption, we conclude that $x$ is unreachable. The facts required to arrive at this conclusion are shown in the implication graph in the figure.

Extracting Dual Transformers We now show how conflict analysis can be viewed as construction of a dual transformer from an implication graph. The dual transformer maps a set of consequences $C$ to the set of antecedents $A$ from which we could have derived $C$. The dual implication transformer defined by an implication graph $I$ is $f_{u} : \mathcal{G}(O) \rightarrow \mathcal{G}(O)$ below.

$$f_{u}(C) \equiv C \cup \{y \mid y \cap m \in C \land (M, m) \in I\}$$

Theorem 16. If the implication graph $I$ represents an overapproximation of $f_o$, the transformer $f_u$ underapproximates $f$.

Proof. Consider an implication graph $I$ that overapproximates the transformer $f_o$ and a downwards-closed set $C \subseteq O$. We prove the inequality $\gamma_{\mathcal{G}(O)} \circ f_{u}(C) \subseteq \gamma_{\mathcal{G}(O)}(C)$.

We show that $f_o(c)$ is in $C$ for all $c \in f_{u}(C)$. Consider $c$ in $f_{u}(C)$. If $c$ is in $C$, we have that $f_o(c)$ is in $C$ because $f_o$ is reductive and $C$ is downwards closed. If $c$ is in $f_{u}(C) \setminus C$, $c$ is, by definition of the form $c = q \cap (\prod M)$, where $q \cap m$ is in $C$ for some $(M, m)$ in $I$. Since $I$ soundly approximates $f_o$, we have that $f_o(\prod M) \sqsubseteq m$, and as a consequence also that $f_o(c) = f_o(q \cap (\prod M)) \sqsubseteq q \cap m$. Since $C$ is downwards closed and $q \cap m$ is in $C$ it holds that $c \in C$. We have shown that $(f_o(c) \mid c \in f_{u}(C))$ is contained in $C$.

The soundness of $f_o$ implies that the downwards-closure of

$$\bigcup \{f(\gamma_{\mathcal{G}(O)}(c)) \mid c \in f_{u}(C)\}$$

is contained in the downwards-closure of

$$\gamma_{\mathcal{G}(O)}(C) \subseteq f_{u}(C).$$

We can rewrite the above condition as $f \circ \gamma_{\mathcal{G}(O)} \circ f_{u} \subseteq \gamma_{\mathcal{G}(O)}$, where $\subseteq$ is the pointwise lifting of the subset order. The functions $(f, \gamma_{\mathcal{G}(O)})$ form a Galois connection, due to which, the inequality $f \circ \gamma_{\mathcal{G}(O)} \circ f_{u} \subseteq \gamma_{\mathcal{G}(O)}$ is equivalent to $\gamma_{\mathcal{G}(O)} \circ f_{u} \subseteq f \circ \gamma_{\mathcal{G}(O)}$. Thus, $f_u$ underapproximates $f$.

Example 4. Consider the interval implication graph $I$ shown in Figure 5, consisting of the following hyperedges:

$$(\{0, y \geq 5\}, (\emptyset, x \leq 2), (\{y \geq 5\}, x \geq 6), (\{x \geq 6, x \leq 2\}, \bot))$$

We compute a least fixed point over $f_{u}$, and represent downwards-closed sets by their maximal elements.

$$C_0 = f_{u}(C_0) = \{x \geq 6, x \leq 2\}$$

$$C_1 = f_{u}(C_1) = C_0 \cup \{y \geq 5, x \leq 2\}, (x \geq 6)\}$$

$$C_2 = f_{u}(C_1) = C_1 \cup \{\top\} = \top$$

It follows that the formula is unsatisfiable. This fixed point computation, in which meet irreducibles are replaced by their explanations, is similar to conflict analysis in SAT solvers.

Generalising Implication Graphs We briefly discuss implication graph generalisation. See [17] for a more algorithmic discussion. In propositional CD\textsubscript{CL}, meet irreducibles are of the form $(x \rightarrow v)$ where $v$ is $t$ or $f$. Meet irreducible partial assignments are incomparable. In more general lattices, such as intervals, there is an order on meet irreducibles. The order on meet irreducibles represents implications that follow from theory axioms. This order
can be used to generalise implication graphs to derive more useful dual transformers. 

**Example 5.** We illustrate implication graph generalisation on the example in Figure 6. Computing \( f_{I_4} \) for the graph \( I \) depicted leads to a sound transformer. Using this transformer one can find, for example, that \( n_2 : b \geq 2 \) is sufficient to deduce a conflict. 

There is a more general graph that still allows deduction of unreachability of \( \delta \): The constraint on the node \( n_2 : b \geq 2 \) can be weakened to obtain \( n_2 : b \geq 1 \). Likewise, \( c_1 : a \leq -42 \) and \( n_1 : a \leq -42 \) can be generalised, respectively, to \( c_1 : T \) and \( n_1 : a \leq -2 \). Computing \( f_{I_4} \) over the resulting graph \( J \) allows one to find that \( n_2 : b \geq 1 \) is sufficient to deduce a conflict. By generalising the graph, we can derive more general conditions under which a location is unreachable.

Learning allows us to negate this condition and discover that the flow graph always satisfies \( n_2 : b \leq 0 \) because the variable \( b \) is in the interval \([-1, -2]\) at the location \( n_2 \). We can now prove that \( \delta \) is unreachable without any assumptions. We note that ACDCCL instantiated in this way over programs is a method to dynamically discover a trace partitioning \([25]\).

**7. Related Work**

A survey of the literature on combining logical solvers and abstract interpreters is beyond the scope of this paper. We limit ourselves to discussing extensions of CDCL solvers to richer problems. 

The DPLL(T) architecture, also called lazy SMT, combines a SAT solver with a solver for the conjunctive fragment of a theory, to determine satisfiability of quantifier-free formulae in a theory \([15]\). In the original lazy SMT architecture, decisions and learning take place in the propositional solver, which may cause theory facts to be enumerated. **Splitting on demand** allows new theory facts to be encoded by propositional variables and added to a formula \([2]\). ACDCCL is an alternative to lazy SMT in which all reasoning happens directly in a fragment of the theory, and no propositional solver is required. We refer to \([3]\) for a theoretical and empirical comparison of ACDCCL and DPLL(T).

Other approaches that jettison a propositional solver in favour of making decisions and learning directly in a theory include natural-domain SMT \([5]\), generalised DPLL \([23]\) and theory-specific solutions for equality \([1]\) and integer linear arithmetic \([20]\) using the cutting-planes proof system. ACDCCL is one recipe for generalising CDCL to new domains. The details of the decision and learning operations have to be designed anew for each theory, but the algorithm used by the solver, its soundness and relative-completeness follow from our work. In separate work, we have instantiated ACDCCL to derive an SMT solver for floating-point logic and shown that this solver performs better than using a propositional encoding \([17]\), and better than DPLL(T) \([17]\).

SMT solvers use the Nelson-Oppen and Delayed Theory Combination methods to combine solvers for different theories. Abstract interpretation is a modular framework, which allows abstract domains to be combined in several different ways \([8]\). The Nelson-Oppen method and DPLL(T) have recently been shown to be instances of more general product constructions in abstract interpretation \([3, 12]\). These product constructions lift to instantiations of ACDCCL: if ACDCCL can be instantiated over domains \( O_1 \) and \( O_2 \), it can be instantiated over the cardinal product \( O_1 \times O_2 \).

Our work allows CDCL to be lifted to program verification problems. We have instantiated ACDCCL with the interval abstract domain to analyse the bounds of variables in programs with floating-point variables \([14]\). Lazy annotation with interpolants \([22]\) lifts CDCL to programs, and uses an SMT solver to implement constraint propagation, and Craig interpolation for learning. We work with abstract domains, and do not assume the domain is closed under Boolean operations, or that it supports interpolation. **Satisfiability Modulo Path Programs** \([18]\) lifts DPLL(T) to programs by combining a SAT solver and an abstract interpreter, and Stålmarck’s method has been lifted to programs in \([28]\).

Our work provides an abstract interpretation view of CDCL, which we extended to other procedures in \([13]\), and to DPLL(T)
in [3]. An account of Stålmarck’s method is given in [27], and [24] contains a transition system view of CDCL.

A natural question is whether ACDCCL is a form of Counter-Example Guided Abstraction Refinement (CEGAR) [4]. CEGAR uses non-deterministic witnesses to construct a new domain and new transformers. ACDCCL implements proof-guided transformer refinement and makes progress when no non-deterministic witness is found. ACDCCL never changes the domain, and this immutability is crucial for efficiency, because the implementations of the abstract domain and transformers can be highly optimised.

8. Conclusion

In this paper, we applied abstract interpretation to study the Conflict Driven Clause Learning algorithm (CDCL) implemented by contemporary SAT solvers. We showed that CDCL can be understood as a lattice-theoretic approximation algorithm that combines over-approximations of greatest fixed points and underapproximations of least fixed points to determine properties of a function on a Boolean lattice. Our generalised Abstract Conflict Driven Clause Learning (ACDCCL) procedure and its correctness proofs rely on properties enjoyed by the data structures in SAT solvers as well as by abstract domains used in practice.

In separate work, we have instantiated ACDCCL to derive an SMT solver and a program analyser and obtained positive results. Problems to investigate in future instantiations include satisfiability in theories of weak arithmetic, assertion checking with relational abstract domains, and nullness of pointers. A second family of problems is to design non-Boolean abstractions of states and traces so that ACDCCL can be applied to model checking problems, and to determine emptiness of non-deterministic automata over finite and infinite words.

We opened the paper by recalling a question posed by Malik and Zhang about whether the lessons from the success of SAT solvers lift to other domains. We believe that our work contributes a mathematical answer to this question that applies to algorithmic issues. Our work does not explain or provide a means to lift the heuristics used by SAT solvers to new problems. We conjecture that heuristics to boost efficiency of constraint propagation are closely related to those for exploiting sparsity in program analysis. It is our hope that future work will provide a framework for understanding these connections and for lifting engineering techniques in SAT solvers to new problem domains.

Acknowledgments

This work was supported by the Toyota Motor Corporation, ERC project 280053, EPSRC project EP/H017585/1, and the FP7 STREP PINCETTE. Vijay D’Silva was supported by a Microsoft Research PhD Scholarship.

We thank the abstract interpretation community, in particular Patrick and Radhia Cousot, for their support and encouragement of this work. Vijay D’Silva thanks Heejong Lee and Wonchan Lee for their diligence in an independent project, which allowed him time for this one.

References