Lifting CDCL to Template-based Abstract Domains for Program Verification*

Rajdeep Mukherjee[1], Peter Schrammel[2], Leopold Haller[3], Daniel Kroening[1], and Tom Melham[1]

[1] University of Oxford, UK
[2] University of Sussex, UK
[3] Google Inc., USA

Abstract. The success of Conflict Driven Clause Learning (CDCL) for Boolean satisfiability has inspired adoption in other domains. We present a novel lifting of CDCL to program analysis called *Abstract Conflict Driven Learning for Programs* (ACDLP). ACDLP alternates between *model search*, which performs over-approximate deduction with constraint propagation, and *conflict analysis*, which performs under-approximate abduction with heuristic choice. We instantiate the model search and conflict analysis algorithms with an abstract domain of *template polyhedra*, strictly generalizing CDCL from the Boolean lattice to a richer lattice structure. Our template polyhedra can express intervals, octagons and restricted polyhedral constraints over program variables. We have implemented ACDLP for automatic bounded safety verification of C programs. We evaluate the performance of our analyser by comparing with CBMC, which uses Boolean CDCL, and Astrée, a commercial abstract interpretation tool. We observe two orders of magnitude reduction in the number of decisions, propagations, and conflicts as well as a 1.5x speedup in runtime compared to CBMC. Compared to Astrée, ACDLP solves twice as many benchmarks and has much higher precision. This is the first instantiation of CDCL with a template polyhedra abstract domain.

1 Introduction

Static program analysis with abstract interpretation [10] is widely used to verify properties of safety-critical systems. Static analyses commonly aim to compute program invariants as fixed-points of abstract transformers. Abstract states are chosen from a lattice that has meet ($\sqcap$) and join ($\sqcup$) operations; the meet precisely models set intersection (or conjunction, taking a logical view), and the join over-approximates set union (or disjunction). Over-approximation in the join operation is one of the sources of precision loss, which can cause false alarms. Typical abstract domains are *non-distributive*; suppose a and b together represent the abstract semantics of a program and c represents a set of abstract behaviours that violate the specification. In a non-distributive domain, $(a \sqcup b) \sqcap c$ can be strictly less precise than $(a \sqcap c) \sqcup (b \sqcap c)$. This means that in typical abstract

* Supported by ERC project 280053 (CPROVER), the H2020 FET OPEN 712689 SC^2 and SRC contracts no. 2012-TJ-2269 and 2016-CT-2707.

domains, analysing program behaviours separately can improve the precision of the analysis. Usual means to address false alarms therefore include not only the use of richer abstract domains, but also of refinements that delay joins or perform some form of case-splitting. Such techniques trade off higher precision against lower efficiency and may be susceptible to case enumeration behaviour.

By contrast, Model Checking (MC) [2] can be seen to operate on distributive lattice structures that represent disjunction without loss of precision. Classical MC directly operates on distributive representations, such as BDDs, while more recent implementations use SAT solvers. SAT solvers themselves operate on partial assignments, which are non-distributive structures. To handle disjunction, case-splitting is performed [15]. Propositional SAT solvers solve large formulae, and are often able to avoid enumerating cases. The impressive performance of modern solvers is credited to well-tuned decision heuristics and sophisticated clause learning algorithms. Collectively, these algorithms are referred to as *Conflict Driven Clause Learning* (CDCL) [3]. An appealing idea is to lift CDCL from the domain of partial assignments to other non-distributive domains.

Abstract Conflict Driven Clause Learning (ACDCL) [13] is one such lattice-based generalization of CDCL. ACDCL is a general algorithmic framework, parameterized by a concrete domain C and an abstract domain A. Classical CDCL can be viewed as an instance of ACDCL in which C is the set of propositional truth assignments and A the domain of propositional partial assignments [17]. Since the concrete domain is a parameter to the framework, ACDCL can in principle be used to build both *logical decision procedures* [5] and *program analyzers*. In the former case, the concrete domain is the set of candidate models for the formula; in the latter case, it is the set of program traces that may lead to an error. Haller et al. [5] pursue the first idea by presenting a floating-point decision procedure that uses interval constraint propagation.

In this paper, we explore the second idea by presenting an extension of ACDCL to program analysis. We call our framework *Abstract Conflict Driven Learning for Programs* (ACDLP). The key insight of ACDLP is to use decisions and learning to reason precisely about disjunctions in non-distributive domains, thereby automatically refining the precision of analysis for safety checking of C programs. We introduce two central components of our framework: an abstract model search algorithm that uses decisions and propagations to search for counterexample trace and an abstract conflict analysis procedure that approximates a set of unsafe traces through transformer learning. We illustrate the application of our framework to program analysis using a *template polyhedra abstract domain* [26], which includes most of the commonly used abstract domains, such as boxes, octagons, zones and TCMs.

We give an experimental evaluation of our analyser compared to CBMC [8], which uses propositional solvers, and to Astrée [4], a commercial abstract interpretation tool. In this paper, we make the following contributions.

1. A novel program analysis framework that lifts model search and conflict analysis procedures of CDCL algorithm over a template polyhedra abstract domain. These techniques are embodied in our tool, *ACDLP*, for automatic bounded safety verification of C programs.
2. A parameterized abstract transformer that guides the model search in forward, backward and multi-way direction for counterexample detection.

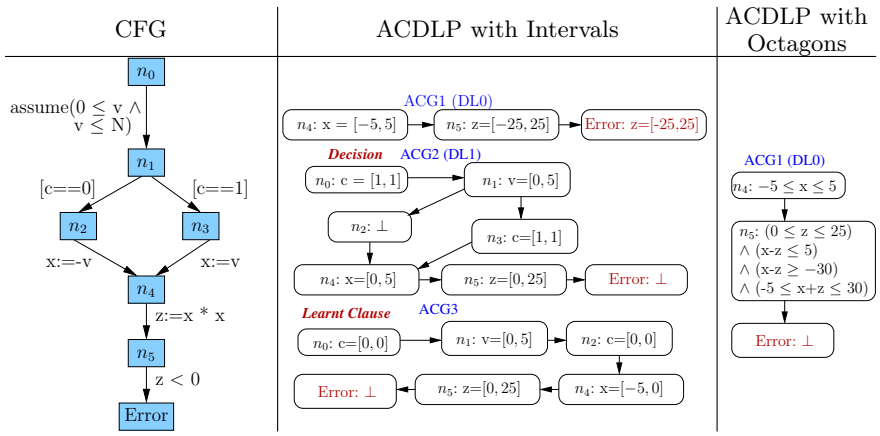

Fig. 1. CFG and corresponding Abstract Conflict Graphs for intervals and octagons

3. A conflict analysis procedure that performs UIP-based transformer learning over template polyhedra abstract domain through abductive reasoning.

2 Motivating Examples

We present two simple examples to demonstrate the essence of ACDLP for bounded verification. For each one, we apply three analysis techniques: *abstract interpretation* (AI), SAT-based *bounded model checking* (BMC) and ACDLP.

First Example. The simple Control-Flow Graph (CFG) in Fig. 1 squares a machine integer and checks whether the result is positive. To avoid overflow, we assume the input v has an upper bound N. This example shows that a) interval analysis in ACDLP is more precise than a forward AI in the interval domain, and b) ACDLP with intervals can achieve a precision similar to that of AI with octagons without employing more sophisticated mechanisms such as trace partitioning [25].

AI versus ACDLP. Conventional forward interval AI is too imprecise to verify safety of this program owing to the control-flow join at node n_4. For example, the state-of-the-art AI tool Astrée requires external hints, provided by manually annotating the code with partition directives at n_1. This tells Astrée to analyse the program paths separately.

ACDLP can be understood as an algorithm to infer such partitions automatically. For the example in Fig. 1, interval analysis with ACDLP is sufficient to prove safety. The analysis records the decisions and deductions in a *trail* data-structure. The trail can be seen to represent a graph structure called the *Abstract Conflict Graph* (ACG) that stores dependencies between decisions and deductions, similar to the way an *Implication Graph* [3] works in a SAT solver. Nodes of the ACG in the second column of Fig. 1 are labelled with the CFG location and the corresponding abstract value. Beginning with the assumption

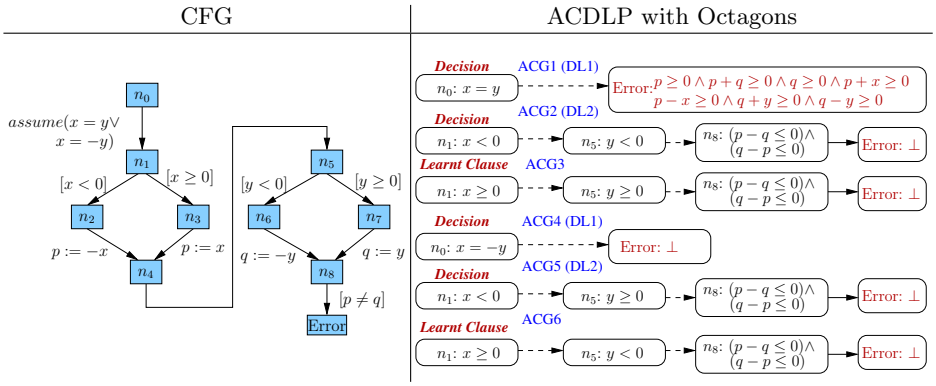

Fig. 2. CFG and corresponding Abstract Conflict Graphs for octagon analysis

that $v = [0, 5]$ at node n_1, the intervals generated by forward analysis in the initial deduction phase at *decision level* 0 (DL0) are $x = [-5, 5]$ and $z = [-25, 25]$. These do not prove safety, as shown in ACG1. So ACDLP makes a heuristic decision, at DL1, to refine the analysis. With the decision $c = [1, 1]$, interval analysis then concludes $x = [0, 5]$ at node n_4, which leads to (Error: $\bot$) in ACG2, indicating that the error location is unreachable and that the program is safe when $c = [1, 1]$.

Reaching (Error: $\bot$) is analogous to reaching a conflict in a propositional SAT solver. At this point, a clause-learning SAT solver learns a reason for the conflict and backtracks to a level such that the learnt clause is *unit*. By a similar process, ACDLP learns that $c = [0, 0]$. That is, all error traces must satisfy $c \neq 1$. The analysis discards all interval constraints that lead to the conflict and backtracks to DL0. ACDLP then performs interval analysis with the learnt clause $c \neq 1$. This also leads to a conflict, as shown in ACG3. The analysis cannot backtrack further and so terminates, proving the program safe. Thus, *decision* and *clause learning* are used to infer the partitions necessary for a precise analysis. Alternatively, the octagon analysis in ACDLP—illustrated in the third column of Fig. 1—can prove safety with propagations only. No decisions are required. Forward AI with octagons in Astrée is also able to prove safety.

Solver	Domain	decisions	propagations	conflicts	conflict literals	restarts
Solver statistics for Fig. 1 (For N = 46000)						
MiniSAT	$BVars \rightarrow \{t, f, ?\}$	233	36436	162	2604	2
ACDLP	$Itvs[NVars]$	1	17	1	1	0
ACDLP	$Octs[NVars]$	0	7	0	0	0
Solver statistics for Fig. 2						
MiniSAT	$BVars \rightarrow \{t, f, ?\}$	4844	32414	570	4750	5
ACDLP	$Octs[NVars]$	4	412	2	2	0

Table 1. SAT-based BMC versus ACDLP for verification of programs in Figs. 1 and 2

Second Example. Fig. 2 shows that octagon analysis in ACDLP is more precise than forward AI in the octagon domain. The CFG in Fig. 2 computes the absolute values of two variables, x and y, under the assumption $(x = y) \vee (x = -y)$.

AI versus ACDLP. Forward AI in the octagon domain infers the octagonal constraint Error: $p \geq 0 \wedge p + q \geq 0 \wedge q \geq 0 \wedge p + x \geq 0 \wedge p - x \geq 0 \wedge q + y \geq 0 \wedge q - y \geq 0$. Clearly this is too imprecise to prove safety. The octagonal analysis in ACDLP is illustrated by the ACGs in Fig. 2. (Due to space limitations, we elide intermediate deductions.) The decision $x = y$ at DL1 is not sufficient to prove safety, as shown in ACG1. So a new decision $x < 0$ is made at DL2, followed by forward propagation that infers $y < 0$ at node n_5. This subsequently leads to safety (Error: $\bot$), as shown in ACG2. The analysis learns the reason for the conflict, discards all deductions in ACG2 and backtracks to DL1. Octagon analysis is run with the learnt constraint $x \geq 0$ and this infers $y \geq 0$ at node n_5, as shown in ACG3. This also leads to safety (Error: $\bot$). The analysis now makes a new decision $x = -y$ at DL1. The procedure is repeated leading to results shown in ACG4, ACG5, and ACG6. Clearly, the decisions $x = -y$ and $x < 0$ also lead to safety. The analysis backtracks to DL0 and returns *safe*. Note that the specific decision heuristic we use in this case exploits the control structure of the program to infer partitions that are sufficient to prove safety.

ACDLP versus BMC. ACDLP can require many fewer iterations than SAT-based BMC due to its ability to reason over much richer lattice structures. A SAT-based BMC converts the program into a bit-vector formula and passes it to a CDCL-based SAT solver for proving safety. Table 1 compares the statistics for BMC with MiniSAT [21] solver to those for interval and octagon analysis in ACDLP. In the column labelled Domains, *BVars* is the set of propositional variables; each of these is mapped to *true* (t), *false* (f) or *unknown* (?). *NVars* is the set of numerical variables; *Itvs*[*NVars*] and *Octs*[*NVars*] are the Interval and Octagon domains over *NVars*. As can be seen, ACDLP outperforms BMC in the total number of *decisions*, *propagations*, *learnt clauses* and *restarts* for both example programs.

3 Program Model and Abstract Domain

3.1 Program Representation

We consider *bounded programs* with safety properties given as a set of assertions, *Assn*, in the program. A bounded program is obtained by a transformation that unfolds loops and recursions a finite number of times. The result is represented by a set $\Sigma = Prog \cup \{\neg \bigwedge_{a \in Assn} a\}$, where *Prog* contains an encoding of the statements in the program as constraints, obtained after translating the program into single static assignment (SSA) form via a data flow analysis. The representation Σ for the program in Fig. 1 is

$$\begin{array}{l} \{g_0 = (0 \leq v \leq N),\ g_1 = (g_0 \wedge c),\ x_0 = v,\ x_1 = -v, \\ \ x_2 = g_1 ? x_0 : x_1,\ g_2 = (g_1 \vee g_0 \wedge \neg c),\ z = x_2 \cdot x_2,\ g_2 \wedge z{<}0\} \end{array} \quad (1)$$

Assignments such as x:=v become equalities $x_1 = v$, where the left-hand side variable gets a subscripted fresh name. Control flow is encoded using guard

Interval	Octagons	Zones	Equality	Fixed-coef. Polyhedra
$a \leq x_i \leq b$	$\pm x_i \pm x_j \leq d$	$x_i - x_j \leq d$	$x_i = x_j$	$a_1x_1 + \ldots + a_nx_n \leq d$

Table 2. Template instances in the template polyhedra domain

variables, e.g. $g_1 = g_0 \wedge c$. Data flow joins become conditional expressions, e.g. $x_3 = g_1?x_1 : x_2$. The assertions in $Assn$ are constraints such as $g_2 \Rightarrow z \geq 0$, meaning that if g_2 holds (i.e., the assertion is reachable), then $z \geq 0$ must hold. We write $Vars$ for the set of variables occurring in Σ. Based on this representation, we define a *safety formula* (φ) as the conjunction of everything in Σ, i.e. $\varphi := \bigwedge_{\sigma \in \Sigma} \sigma$. The formula φ is unsatisfiable if and only if the program is safe.

3.2 Abstract Domain

In this paper, we instantiate ACDLP over a reduced product domain [11] $D[Vars] = \mathcal{B}^{|BVars|} \times \mathcal{TP}[NVars]$ where $\mathcal{B}$ is the Boolean domain that permits abstract values $\{\mathsf{true}, \mathsf{false}, \bot, \top\}$ over boolean variables $BVars$ in the program, and $\mathcal{TP}$ is a *template polyhedra* [26] domain over the numerical (bit-vector) variables $NVars$. Our template polyhedra domain can express various relational and non-relational templates over $NVars$, as given in Table 2.

Template Polyhedra Abstract Domain. An abstract value of the template polyhedra domain [26] represents a set X of values of the vector $\boldsymbol{x}$ of numerical (bit-vector) variables $NVars$ of their respective data types. (Currently, signed and unsigned integers are supported.) For example, in the program given by Eq. (1), we have four numerical variables, written as the vector $\boldsymbol{x} = (x_0, x_1, x_2, z)$. An abstract value is a constant vector $\boldsymbol{d}$ that represents sets of values for $\boldsymbol{x}$ for which $\boldsymbol{C}\boldsymbol{x} \leq \boldsymbol{d}$, for a fixed coefficient matrix $\boldsymbol{C}$. The domain containing $\boldsymbol{d}$ is augmented by a special element $\bot$ to denote the minimal element of the lattice. There are several optimisation techniques [26] for computing the domain operations, such as meet ($\sqcap$) and join ($\sqcup$), in the template polyhedra domain. In our implementation, we use the strategy iteration approach of [6]. The abstraction function is $\alpha(X) = min\{\boldsymbol{d} \mid \boldsymbol{C}\boldsymbol{x} \leq \boldsymbol{d}, \boldsymbol{x} \in X\}$, where min is applied component-wise. The concretisation $\gamma(\boldsymbol{d})$ is the set $\{\boldsymbol{x} \mid \boldsymbol{C}\boldsymbol{x} \leq \boldsymbol{d}\}$ and $\gamma(\bot) = \emptyset$, i.e., the empty polyhedron.

For notational convenience we will use conjunctions of linear inequalities, for example $x_1 \geq 0 \wedge x_1 - z \leq 30$, to write the abstract domain value $\boldsymbol{d} = \begin{pmatrix} 0 \\ 30 \end{pmatrix}$, with $\boldsymbol{C} = \begin{pmatrix} -1 & 0 \\ 1 & -1 \end{pmatrix}$ and $\boldsymbol{x} = \begin{pmatrix} x_1 \\ z \end{pmatrix}$; true corresponds to abstract value $\top$ and false to abstract value $\bot$. For a program with $N = |NVars|$ variables, the template matrix $\boldsymbol{C}$ for the interval domain $Itvs[NVars]$, has $2N$ rows. Hence, it generates at most $2N$ inequalities, one for the upper and lower bounds of each variable. For octagons $Octs[NVars]$, we have at most $2N^2$ inequalities, one for the upper and lower bounds of each variable and sums and differences for each pair of variables. Unlike a non-relational domain, a relational domain such

as octagons requires the computation of a *closure* to obtain a normal form, necessary for precise domain operation. The closure computes all implied domain constraints. An example of a closure computation for octagonal inequalities is $\mathit{closure}((x - y \leq 4) \wedge (y - z \leq 5)) = ((x - y \leq 4) \wedge (y - z \leq 5) \wedge (x - z \leq 9))$. For octagons, closure is the most critical and expensive operator; it has cubic complexity in the number of program variables. We therefore compute closure lazily in template polyhedra domain in our abstract model search procedure, which is described in Section 5.3.

Abstract Transformers. An abstract transformer $[\![\sigma]\!]_D$ transforms an abstract value a through a constraint σ; it *deduces* information from a and σ. The best transformer is

$$[\![\sigma]\!]_D(a) = a \sqcap \alpha(\{u \mid u \in \gamma(a), u \models \sigma\}) \tag{2}$$

where we write $u \models \sigma$ if the concrete value u satisfies the constraint σ. Any abstract transformer that over-approximates the best abstract transformer is a sound transformer and can be used in our algorithm. For example, we can deduce $[\![x = 2(y + z)]\!]_D(a) = (0 \leq y \leq 2 \wedge 1 \leq y - z \leq 1 \wedge -2 \leq x \leq 6)$ for the abstract value $a = (0 \leq y \leq 2 \wedge 1 \leq y - z \leq 1)$. We denote the set of abstract transformers for a safety formula φ using the abstract domain D by $\mathcal{A} = \{[\![\sigma]\!]_D \mid \sigma \in \Sigma\}$.

3.3 Precise Complementation in Abstract Domains

An important property of a clause-learning SAT solver is that each non-singleton element of the partial assignment domain can be decomposed into a set of *precisely complementable* singleton elements [13]. This property is necessary to learn elements that guide the model search away from the conflicting region of the search space. Most numerical abstract domains, such as intervals and octagons, lack complements in general: not every domain element has a precise complement. But these domain elements can be represented as intersections of half-spaces, each of which admits a precise complement. We formalise this in the sequel.

Definition 1. *A* meet irreducible m *in a complete lattice structure* A *is an element with the following property.*

$$\forall m_1, m_2 \in A : m_1 \sqcap m_2 = m \implies (m = m_1 \vee m = m_2), m \neq \top \tag{3}$$

The meet irreducibles in the Boolean domain $\mathcal{B}$ for a variable x are x and $\neg x$. The meet irreducibles in the template polyhedra domain are all elements that concretise to half-spaces; i.e., they can be represented by a single inequality. For the interval domain, these are $x \leq d$ or $x \geq d$ for constants d.

Definition 2. *A* meet decomposition $\mathit{decomp}(a)$ *of an abstract element* $a \in D$ *is a set of meet irreducibles* $M \subseteq D$ *such that* $a = \bigsqcap_{m \in M} m$.

For polyhedra this means that each polyhedron can be written as an intersection of half-spaces. For example, the meet decomposition of the interval domain element $\mathit{decomp}(2 \leq x \leq 4 \wedge 3 \leq y \leq 5)$ is the set $\{x \geq 2, x \leq 4, y \geq 3, y \leq 5\}$.

Definition 3. *An element* $a \in D$ *is called* precisely complementable *iff there exists* $\bar{a} \in D$ *such that* $\neg\gamma(\bar{a}) = \gamma(a)$. *That is, there is an element whose complemented concretisation equals the concretisation of* a.

The precise complementation property of a partial assignment lattice can be generalised to other lattice structures. For example, the precise complement of a meet irreducible ($x \leq 2$) in the interval domain over integers is ($x \geq 3$), or the precise complement of the meet irreducible ($x + y \leq 1$) in the octagon domain over integers is ($x + y \geq 2$). Our domain implementation supports a precise complementation operation. Standard abstract interpretation does not require a complementation operator, so abstract domain libraries, such as APRON [19], do not provide it. But it can be implemented with the help of a meet decomposition as explained above.

4 Abstract Conflict Driven Learning for Programs

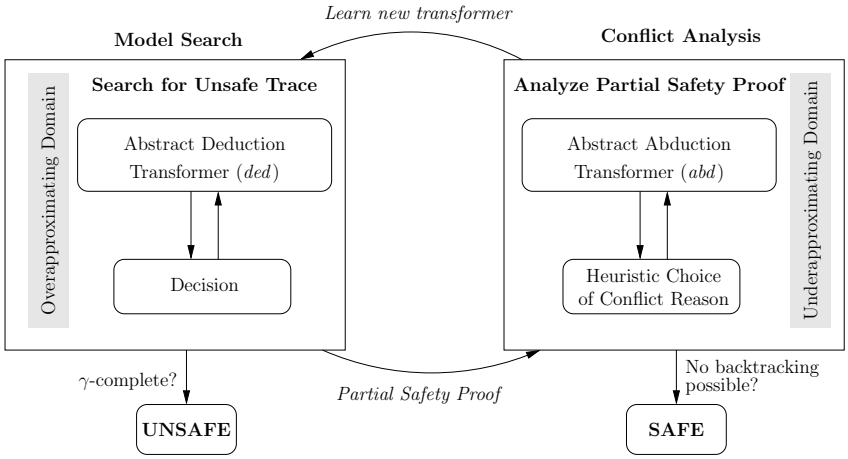

Fig. 3. Architectural view of ACDLP

Figure 3 presents our framework called *Abstract Conflict Driven Learning for Programs* that uses abstract model search and abstract conflict analysis procedures for safety verification of C programs. The model search procedure operates on an over-approximate domain of program traces through repeated application of abstract deduction transformer, *ded*, and decisions in order to search for a counterexample trace. If the model search finds a satisfying assignment (corresponding deduction transformer is γ-complete), then ACDLP terminates with a counterexample trace, and the program is *unsafe*. Else, if a conflict is encountered, then it implies that the corresponding program trace is either not valid or safe. ACDLP then moves to the conflict analysis phase where it learns the reason for the conflict from partial safety proof using an abstract abductive transformer, *abd*, followed by a heuristic choice of conflict reason. Similar to a SAT solver, ACDLP picks one conflict reason from multiple incomparable reasons for conflict for efficiency reasons. Hence, it operates over an under-approximate domain of conflict reasons. A conflict reason under-approximates a set of invalid or safe

Algorithm 1: Abstract Conflict Driven Learning $ACDLP_{H_P,H_D,H_C}(\mathcal{A})$

```
   input  : A program in the form of a set of abstract transformers 𝒜.
   output : The status safe or unsafe.
 1 𝒯 ← ⟨⟩, ℛ ← []
 2 result ← deduce_{H_P}(𝒜, 𝒯, ℛ)
 3 if result = conflict then return safe
 4 while true do
 5 |   if result = sat then return unsafe
 6 |   q ← decide_{H_D}(abs(𝒯))
 7 |   𝒯 ← 𝒯 · q
 8 |   ℛ[|𝒯|] ← ⊤
 9 |   result ← deduce_{H_P}(𝒜, 𝒯, ℛ)
10 |   do
11 |   |   if ¬analyzeConflict_{H_C}(𝒜, 𝒯, ℛ) then return safe
12 |   |   result ← deduce_{H_P}(𝒜, 𝒯, ℛ)
13 |   while result = conflict
14 end
```

traces. The conflict analysis returns a learnt transformer (negation of conflict reason) that over-approximates a set of valid and unsafe traces. Model search is repeated with this new transformer. Else, if no further backtracking is possible, then ACDLP terminates and returns *safe*. We present the ACDLP algorithm in the subsequent section.

The input to ACDLP (Algorithm 1) is a program in the form of a set of abstract transformers $\mathcal{A} = \{[\![\sigma]\!]_D | \sigma \in \Sigma\}$ w.r.t. an abstract domain D. Recall that the safety formula $\bigwedge_{\sigma\in\Sigma} \sigma$ is unsatisfiable if and only if the program is safe. The algorithm is parametrised by heuristics for propagation (H_P), decisions (H_D), and conflict analysis (H_C). The algorithm maintains a propagation trail $\mathcal{T}$ and a reason trail $\mathcal{R}$. The propagation trail stores all meet irreducibles inferred by the abstract model search phase (deductions and decisions). The reason trail maps the elements of the propagation trail to the transformers $ded \in \mathcal{A}$ that were used to derive them.

Definition 4. *The* abstract value $abs(\mathcal{T})$ *corresponding to the propagation trail* $\mathcal{T}$ *is the conjunction of the meet irreducibles on the trail:* $abs(\mathcal{T}) = \bigsqcap_{m\in\mathcal{T}} m$ *with* $abs(\mathcal{T}) = \top$ *if* $\mathcal{T}$ *is the empty sequence.*

The algorithm begins with an empty $\mathcal{T}$, an empty $\mathcal{R}$, and the abstract value $\top$. The procedure *deduce* (details in Section 5) computes a greatest fixed-point over the transformers in $\mathcal{A}$ that refines the abstract value, similar to the Boolean Constraint Propagation step in SAT solvers. If the result of *deduce* is conflict ($\bot$), the algorithm terminates with safe. Otherwise, the analysis enters into the while loop at line 4 and makes a new decision by a call to *decide* (see Section 5.4), which returns a new meet irreducible q. We append q to the trail $\mathcal{T}$. The decision q refines the current abstract value $abs(\mathcal{T})$ represented by the trail, i.e., $abs(\mathcal{T}\cdot q) \sqsubseteq abs(\mathcal{T})$. For example, a decision in the interval domain restricts the range of intervals for variables. We set the corresponding entry in the reason trail $\mathcal{R}$ to $\top$ to mark it as a decision. Here, the index of $\mathcal{R}$ is the size of trail $\mathcal{T}$, denoted

Algorithm 2: Abstract Model Search $deduce_{H_P}(\mathcal{A}, \mathcal{T}, \mathcal{R})$

```
   input  : A program in the form of a set of abstract transformers 𝒜, a
            propagation trail 𝒯, and a reason trail ℛ.
   output : sat or conflict or unknown
 1 worklist ← initWorklist_{H_P}(𝒜)
 2 while !worklist.empty() do
 3 |   ded^L ← worklist.pop()
 4 |   a ← ded^L(abs(𝒯))
 5 |   if a = ⊥ then
 6 |   |   ℛ[⊥] ← ded^L
 7 |   |   worklist.clear()
 8 |   |   return conflict
 9 |   else
10 |   |   v = onlyNew(a)
11 |   |   𝒯 ← 𝒯 · decomp(v)
12 |   |   ℛ[|𝒯|] ← ded^L
13 |   |   updateWorklist_{H_P}(worklist, v, ded^L, 𝒜)
14 end
15 if 𝒜 is γ-complete at abs(𝒯) then return sat
16 return unknown
```

by $|\mathcal{T}|$. The procedure *deduce* is called next to infer new meet irreducibles based on the current decision. The model search phase alternates between the decision and deduction until *deduce* returns either sat or conflict.

If *deduce* returns sat, then we have found an abstract value that represents models of the safety formula, which are counterexamples to the required safety property, and so ACDLP returns unsafe. If *deduce* returns conflict, the algorithm enters in the *analyzeConflict* phase (see Section 6) to learn the reason for the conflict. There can be multiple incomparable reasons for conflict. ACDLP heuristically chooses one reason C and learns it by adding it as an abstract transformer to $\mathcal{A}$. The analysis backtracks by removing the content of $\mathcal{T}$ up to a point where it does not conflict with C. ACDLP then performs deductions with the learnt transformer. If *analyzeConflict* returns false, then no further backtracking is possible. Thus, the safety formula is unsatisfiable and ACDLP returns safe. An example demonstrating step-by-step execution of the ACDLP algorithm is available at [23].

5 Abstract Model Search for Template Polyhedra

Model search in a SAT solver has two steps: *deductions*, which are repeated application of the unit rule (also called Boolean Constraint Propagation, or BCP), to refine current partial assignments, and *decisions* to heuristically guess a value for an unassigned literal. BCP can be seen to compute the greatest fixed point over the partial assignment domain [13]. Below, we present an abstract model search procedure that computes a greatest fixed point over abstract transformers $[\![\sigma]\!]_D$.

5.1 Parametrised Abstract Transformers

The key considerations for an abstract transformer are precision and efficiency. A precise transformer is usually less efficient than a more imprecise one. In this paper, we present a specialised variant of the abstract transformer to compute deductions called *Abstract Deduction Transformer* (ADT), which is parametrised by a given *subdomain* $L \subseteq D$. A subdomain contains a chosen subset of the elements in D including $\bot$ and $\top$ that forms a lattice. The use of a subdomain serves two purposes: a) It allows us elegantly and flexibly to guide the deductions in *forward*, *backward* or *multi-way* direction, which in turn affects the analysis precision, and b) it makes deductions more efficient, for example by performing lazy closure in template polyhedra domain.

An ADT is defined formally as follows.

$$[\![\sigma]\!]^L_D(a) = a \sqcap_D \alpha_L(\{u \mid u \in \gamma_D(a), u \models \sigma\}) \tag{4}$$

For $L = D$, the ADT is identical to the abstract transformer defined in Eq. (2) in Section 3. Note that a restricted subdomain makes a transformer less precise but more efficient. Conversely, an unrestricted subdomain make a transformer more precise, but less efficient. Therefore, we have the property $[\![\sigma]\!]^D_D(a) \sqsubseteq [\![\sigma]\!]^L_D(a)$. To illustrate point (1), we give examples that demonstrate how the choice of subdomain influences the propagation direction:

Forward Transformer. For an abstract value $a = (0 \leq y \leq 1 \wedge 5 \leq z)$, $\sigma = (x = y + z)$, and $L = \mathit{Itvs}[\{x\}]$, we have $[\![x = y + z]\!]^{\mathit{Itvs}[\{x\}]}_{\mathit{Itvs}[\{x,y,z\}]}(a) = a \sqcap (x \geq 6)$. Assuming that the equality $x = y + z$ originated from an assignment to x, this performs a right-hand side (rhs) to left-hand side (lhs) propagation and hence emulates a forward analysis.

Backward Transformer. For an abstract value $a = (0 \leq x \leq 10 \wedge 0 \leq y \leq 1 \wedge 5 \leq z)$, $\sigma = (x = y + z)$, and $L = \mathit{Itvs}[\{y, z\}]$, we have $[\![x = y + z]\!]^{\mathit{Itvs}[\{y,z\}]}_{\mathit{Itvs}[\{x,y,z\}]} = a \sqcap (z \leq 10)$. This performs an lhs-to-rhs propagation and hence emulates a backward analysis.

Multi-way Transformer. For an abstract value $a = (c \leq 1 \wedge c \geq 1 \wedge x \leq 5 \wedge x \geq 5)$, $\sigma = ((c = (x = y)) \wedge y = y + 1)$ and $L = \mathit{Itvs}[\{c, x, y\}]$, we have $[\![\sigma]\!]^{\mathit{Itvs}[\{c,x,y\}]}_{\mathit{Itvs}[\{c,x,y\}]} = a \sqcap (y \leq 6 \wedge y \geq 6)$. This performs an lhs-to-rhs propagation for $c = (x = y)$ and rhs to lhs propagation for $y = y + 1$ and hence emulates a multi-way analysis.

5.2 Algorithm for the Deduction Phase

Algorithm 2 presents the deduction phase *deduce* in our abstract model search procedure. The input to *deduce* is the set of abstract transformers, a propagation trail ($\mathfrak{T}$) and a reason trail ($\mathfrak{R}$). Additionally, the procedure *deduce* is parametrised by a propagation heuristic (H_P). We write the ADT $[\![\sigma]\!]^L_D$ as ded^L in Algorithm 2. The algorithm maintains a *worklist*, which is a queue that contains ADTs. The propagation heuristics provides two functions *initWorklist*

and *updateWorklist*. The order of the elements in the worklist and the subdomain L associated with each ADT (ded^L) determine the propagation strategy (forward, backward, multi-way). These two functions construct a subdomain (L) for ded^L by calling the function *MakeL* such that $L = MakeL_D(V)$, where V are the variables that appear in ded^L. The abstract value a is updated upon the application of ded^L in line 4 in Algorithm 2. The function $onlyNew(a) = \bigsqcap(decomp(a) \setminus decomp(abs(\mathcal{T})))$ is used to filter out all meet irreducibles that are already on the trail in order to obtain only new deductions (v) when applying the ADT (shown in line 10). Depending on the propagation heuristics, *updateWorklist* adds ADTs ded^L to the worklist that contain variables that appear in v, and updates the subdomains of the ADTs in the worklist to include the variables in v (shown in line 13).

If ded^L deduces $\bot$, then the procedure *deduce* returns conflict (shown in line 8). Otherwise, when a fixed-point is reached, i.e., the worklist is empty, we check whether the abstract transformers $\mathcal{A}$ are γ-complete [13] for the current abstract value $abs(\mathcal{T})$ (shown in line 15). Intuitively, this checks whether all concrete values in $\gamma(abs(\mathcal{T}))$ satisfy the safety formula φ, where $\varphi := \bigwedge_{\sigma \in \Sigma} \sigma$ is obtained from the program transformation (as defined in Section 3.1). If it is indeed γ-complete, then *deduce* returns sat. Otherwise, the algorithm returns unknown and ACDLP makes a new decision.

5.3 Computing Lazy Closure for Template Polyhedra

An advantage of our formalism in Eq. (4) is that the *closure* operation for relational domains can be computed in a lazy manner through the construction of a subdomain, L. The construction of L allows us to perform one step of the closure operation when ded^L is applied. For example, let us consider $D = Octs[\{x, y, z\}]$ and $V = \{y\}$. An octagonal inequality relates at most two variables. Thus it is sufficient to consider the subdomain $MakeL_D(\{y\}) = Octs[\{y\}] \cup Octs[\{x, y\}] \cup Octs[\{y, z\}]$, which will compute the one-step transitive relations of y with each of the other variables. Only if any subsequent abstract deduction transformer makes new deductions on x or z, then the next step of the closure will be computed through the subdomain $Octs[\{x, z\}]$. Hence, an application of each abstract deduction transformer does not compute the full closure in the full domain, but compute only a single step of the closure in a subdomain. This makes each deduction step more efficient but may require more steps to reach the fixed point.

5.4 Decisions

A decision q is a meet irreducible that refines the current abstract value $abs(\mathcal{T})$, when the result of the fixed-point computation through deduction is neither a *conflict* nor a *satisfiable model* of φ. A decision must always be consistent with respect to the trail $\mathcal{T}$, i.e., $abs(\mathcal{T} \cdot q) \neq \bot$. A new decision increases the decision level by one. Given the current abstract value $abs(\mathcal{T})$, the procedure *decide* in Algorithm 1 heuristically returns a meet irreducible.

For example, a decision in the interval domain can be of the form xRd where $R \in \{\leq, \geq\}$, and d is the bound. A decision in the octagon domain can specify

relations between variables, and can be of the form $ax - by \leq d$, where x and y are variables, $a, b \in \{-1, 0, 1\}$ are coefficients, and d is a constant. The detailed description of the different decision heuristics in ACDLP is available at [23].

6 Abstract Conflict Analysis for Template Polyhedra

Propositional conflict analysis with FIRST-UIP [3] can be seen as abductive reasoning that under-approximates a set of models that do not satisfy a formula [13, 15]. Thus, we view abduction as De Morgan dual of deduction whose result does not need to be consistent with respect to a background theory. Below, we present an abstract conflict analysis procedure, *analyzeConflict* of Algorithm 1, that uses a domain-specific abductive transformer for effective learning. A conflict analysis procedure involves two steps: *abduction* and *heuristic choice for generalisation*. Abduction infers possible generalised reasons for a conflict, which is followed by heuristically selecting a generalisation. Below, we define a global conflict transformer that gives a set of models that do not satisfy a formula.

Definition 5. *Given a formula φ, a downwards closed set of abstract elements Q, and domain D, $conf_\varphi^D(Q) = \{u \in D \mid u \in Q \vee u \not\models \varphi\}$, that is, it returns the set of abstract models that do not satisfy φ or are approximated by Q. An abstract abductive transformer, $abd_\varphi^D(Q)$, corresponds to the under-approximation of the global conflict transformer, $conf_\varphi^D(Q)$.*

For example, given a formula $\varphi = (x = y + 1 \ \wedge \ x \geq 0)$ and an interval abstract element $Q = (y \leq -5 \wedge x \leq -4)$, $conf_\varphi^{Itvs}(Q) = \{(y \leq -5 \wedge x \leq -4)$, $(y \leq -2 \wedge x < 0), \ldots, (y \leq 2 \wedge x \leq 10), \ldots\}$. Now, an abstract abductive transformer for φ is given by $abd_\varphi^{Itvs}(Q) = (y \leq -2 \wedge x < 0)$, which clearly underapproximates $conf_\varphi^{Itvs}$ as well as strictly generalizes the reason for Q.

The main idea of abductive reasoning is to iteratively replace an abstract element s in the conflict reason by a partial assignment that is sufficient to infer s. Conflict abduction is performed by obtaining cuts through markings in the trail $\mathcal{T}$ using an abstract Unique Implication Point (UIP) search algorithm [3]. Every cut is a reason for a conflict. The UIP search can be understood as graph cutting in an Abstract Conflict Graph, which is defined next.

Definition 6. *An Abstract Conflict Graph (ACG) is a directed acyclic graph in which the vertices are defined by deduced elements (including a special conflict node ($\perp$)) or a decision node in the trail $\mathcal{T}$. The edges in ACG are obtained from the reason trail $\mathcal{R}$ that maps pairs of elements in $\mathcal{T}$ to the abstract transformers that are used to derive the deduced elements.*

Abstract UIP Search An abstract UIP is a node in the ACG that must be traversed on every path between a decision node and the conflict. An abstract UIP cut is necessary to ensure that the learnt clauses are asserting after backtracking and prevent cyclic algorithm behavior. An abstract UIP algorithm [5] traverses the trail $\mathcal{T}$ starting from the conflict node and computes a cut that suffices to produce a conflict. For example, consider a formula $\varphi :=$

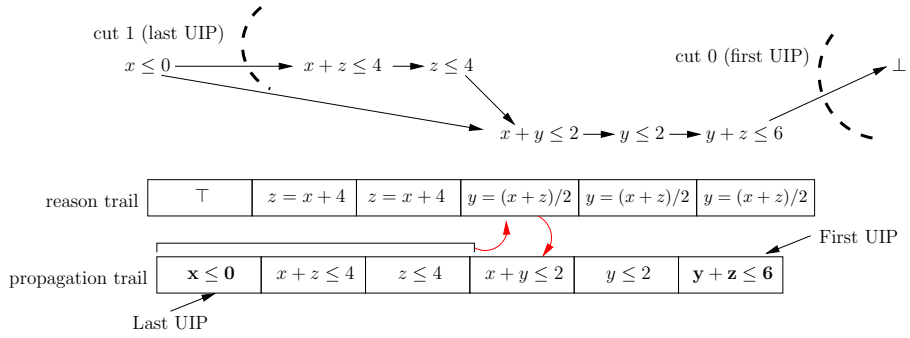

Fig. 4. Finding the Abstract UIP in the octagon domain

($x{+}4{=}z \wedge x{+}z{=}2y \wedge z{+}y{>}10$). As before, the trail can be viewed to represent an ACG, given in Fig. 4, that records the sequence of deductions in the octagon domain that are inferred from a decision $x{\leq}0$ for the formula φ. The arrows (in red) indicate the relationship between the reason trail and propagation trail at the bottom of Fig. 4. For the partial abstract value, $a = (x \leq 0 \wedge x + z \leq 4 \wedge z \leq 4)$, obtained from the trail, the result of the abstract deduction transformer is $[\![y = (x + z)/2]\!]_{Octs}(a) = (x + y \leq 2 \wedge y \leq 2 \wedge y + z \leq 6)$. A conflict ($\bot$) is reached for the decision $x{\leq}0$. Note that there exist multiple incomparable reasons for the conflict, marked as *cut 0* and *cut 1* in Fig. 4. Here, cut 0 is the first UIP (node closest to conflict node). Choosing cut 0 yields a learnt clause $y + z > 6$, which is obtained by negating the reason for the conflict. The abstract UIP algorithm returns a learnt transformer $AUnit$, which is described next.

Learning in Template Polyhedra Domain Learning in a propositional solvers yields an asserting clause [3] that expresses the negation of the conflict reasons. We present a lattice-theoretic generalisation of the *unit rule* for template-based abstract domains that learns a new transformer called *abstract unit transformer* ($AUnit$). We add $AUnit$ to the set of abstract transformers $\mathcal{A}$. $AUnit$ is a generalisation of the propositional unit rule to numerical domains. For an abstract lattice D with complementable meet irreducibles and a set of meet irreducibles $C \subseteq D$ such that $\bigsqcap C$ does not satisfy φ, $AUnit_C : D \to D$ is formally defined as follows.

$$AUnit_C(a) = \begin{cases} \bot & \text{if } a \sqsubseteq \bigsqcap C & (1) \\ \bar{t} & \text{if } t \in C \text{ and } \forall t' \in C \setminus \{t\}. a \sqsubseteq t' & (2) \\ \top & \text{otherwise} & (3) \end{cases}$$

Rule (1) shows $AUnit$ returns $\bot$ when $a \sqsubseteq \bigsqcap C$ is conflicting. Rule (2) of $AUnit$ infers a valid meet irreducible, which implies that C is unit. Rule (3) of $AUnit$ returns $\top$ which implies that the learnt clause is not *asserting* after backtracking. This would prevent any new deductions from the learnt clause. Progress is then made by decisions. An example of $AUnit$ for $C = \{x \geq 2, x \leq 5, y \leq 7\}$ is below. Rule 1: For $a = (x \geq 3 \wedge x \leq 4 \wedge y \geq 5 \wedge y \leq 6)$, $AUnit_C(a) = \bot$, since $a \sqsubseteq \bigsqcap C$. Rule 2: For $a = (x \geq 3 \wedge x \leq 4)$, $AUnit_C(a) = (y \geq 8)$, since $a \sqsubseteq (2 \leq x \leq 5)$.

Rule 3: For $a = (x \geq 1 \wedge y \leq 10)$, $AUnit_C(a) = \top$.

Backjumping A backjumping procedure removes all the meet irreducibles from the trail up to a decision level that restores the analysis to a non-conflicting state. The backjumping level is defined by the meet irreducibles of the conflict clause that is closest to the root (decision level 0) where the conflict clause is still unit. If a conflict clause is globally unit, then the backjumping level is the root of the search tree and *analyzeConflict* returns false, otherwise it returns true.

7 Experimental Results

We have implemented ACDLP for bounded safety verification of C programs. ACDLP is implemented in C++ on top of the CPROVER [12] framework as an extension of 2LS [27] and consists of around 9 KLOC. The template polyhedra domain is implemented in C++ in 10 KLOC. Templates can be intervals, octagons, zones, equalities, or restricted polyhedra. Our domain handles all C operators, including bit-wise ones, and supports precise complementation of meet irreducibles, which is necessary for conflict-driven learning. Our tool and benchmarks are available at `http://www.cprover.org/acdcl/`.

We verified a total of 85 ANSI-C benchmarks. These are derived from: (1) the bit-vector regression category in SV-COMP'16; (2) ANSI-C models of hardware circuits auto-generated by v2c [24] from VIS Verilog models and opencores.org; (3) controller code with varying loop bounds auto-generated from Simulink model and control intensive programs with nested loops containing relational properties. All the programs with bounded loops are completely unrolled before analysis.

We compare ACDLP with the state-of-the-art SAT-based bounded model checker CBMC ([7], version 5.5) and a commercial static analysis tool, Astrée ([1], version 14.10). CBMC uses MiniSAT 2.2.1 in the backend. Astrée uses a range of abstract domains, which includes interval, bit-field, congruence, trace partitioning, and relational domains (octagons, polyhedra, zones, equalities, filter). To enable fair comparison using Astrée, all bounded loops in the program are completely unwound up to a given bound before passing to Astrée. This prevents Astrée from widening loops. ACDLP is instantiated to a product of the Booleans and the Interval or Octagon domain. ACDLP is also configured with a decision heuristic (ordered, random, activity-based), propagation (forward, backward and multi-way), and conflict-analysis (learning UIP, DPLL-style). The timeout for our experiments is set to 200 seconds.

ACDLP versus CBMC Fig. 5 presents a comparison between CBMC and ACDLP. Fig. 5(a) clearly shows that the SAT-based analysis makes significantly more decisions than ACDLP for all the benchmarks. The points on the extreme right below the diagonal in Fig. 5(b) show that the number of propagations in the SAT-based analysis is maximal for benchmarks that exhibit relational behaviour. These benchmarks are solved by the octagon domain in ACDLP. We see a reduction of at least two orders of magnitude in the total number of decisions, propagations and conflicts compared to analysis using CBMC.

Out of 85 benchmarks, SAT-based analysis could prove only 26 benchmarks without any restarts. The solver was restarted in the other 59 cases to avoid

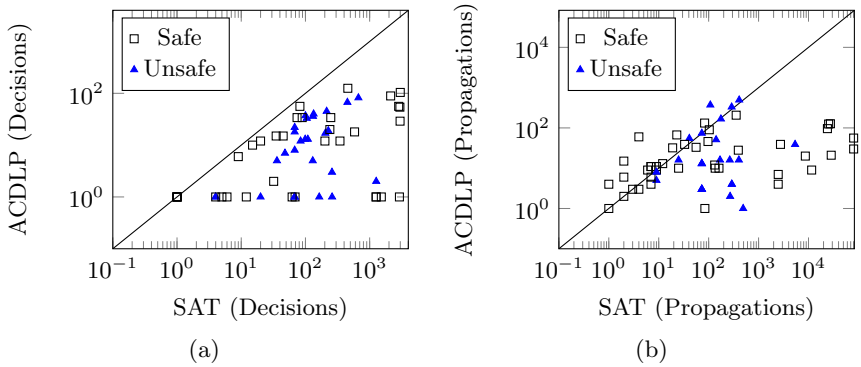

Fig. 5. Comparing SAT-based BMC and ACDLP: number of decisions and propagations

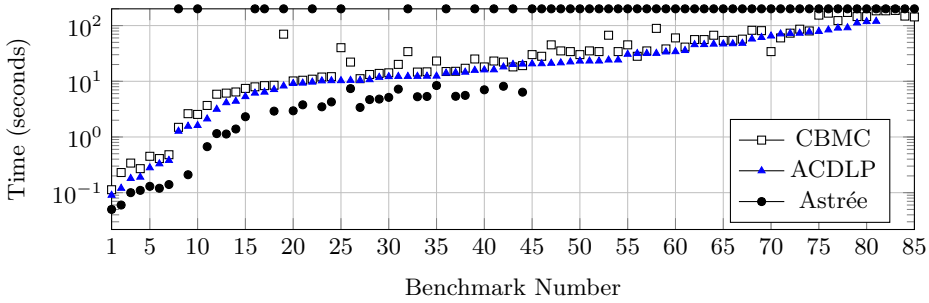

Fig. 6. Runtime comparison between CBMC, Astrée and ACDLP

spending too much time in "hopeless" branches. By contrast, ACDLP solved all 85 benchmarks without restarts. The runtime comparison between ACDLP and CBMC is shown in Figure 6. ACDLP is 1.5X faster than CBMC. The superior performance of ACDLP is attributed to the decision heuristics, which exploit the high-level structure of the program, combined with the precise deduction by multi-way transformer and stronger learnt clauses aided by the abstract domains.

ACDLP versus Astrée To enable precise analysis with Astrée, we manually instrument the benchmarks with partition directives `__ASTREE_partition_control` at various control-flow joins. These directives provide external hints to Astrée to guide its internal trace partitioning domain. Figure 6 demonstrates that Astrée is 2X faster than ACDLP for 37% cases (32 out of 85); but the analysis using Astrée shows a high degree of imprecision (marked as timeout in Figure 6). Astrée reported 53 false alarms among 85 benchmarks. By contrast, the analysis using ACDLP produces correct results for 81 benchmarks. ACDLP times out for 4 benchmarks. Clearly, ACDLP has higher precision than Astrée. A detailed comparison between ACDLP, CBMC and Astrée is available at [23].

Our experimental evaluation suggests that ACDLP can be seen as a technique to improve the efficiency of SAT-based BMC. Additionally, ACDLP can

also be perceived as an automatic way to improve the precision of conventional abstract interpretation over non-distributive lattices through automatic partitioning techniques such as decisions and transformer learning.

8 Related Work

Fränzle et al. [16] present a tight integration of SAT solving with interval-based constraint solving to handle large constraint systems. Silva et al. [15] present an abstract interpretation account of satisfiability algorithms derived from DPLL procedures. The work of [14] is a very early instantiation of abstract CDCL [15] as an interval-based decision procedure for programs, but in a purely logical setting. A similar technique that lifts DPLL(T) to programs is Satisfiability Modulo Path Programs (SMPP) [18]. SMPP enumerates program paths using a SAT formula, which are then verified using abstract interpretation.

The lifting of CDCL to first-order theories is proposed in [9, 20, 22]. Unlike previous work that operates on a fixed first-order lattice, ACDLP can be instantiated with different abstract domains as well as product domains. This involves model search and learning in abstract lattices. A similar technique that lifts decisions, propagations and learning to theory variables is Model-Constructing Satisfiability Calculus (mcSAT) [22].

ACDLP is not, however, similar to abstraction refinement. ACDLP works on a fixed abstraction. Also, transformer learning in ACDLP does not soundly over-approximate the existing program transformers. Hence, transformer learning in ACDLP is distinct from transformer refinement in classical CEGAR.

9 Conclusions

We present a general algorithmic framework for lifting the model search and conflict analysis procedures in DPLL-style satisfiability solvers to program analysis. We embody these techniques in a tool, ACDLP, for automatic bounded safety verification of C programs over a template polyhedra abstract domain.

We present an *abstract model search* procedure that uses a parameterised abstract transformer to flexibly control the precision and efficiency of the deductions in the template polyhedra abstract domain. The underlying expressivity of the abstract domain helps our decision heuristics to exploit the high-level structure of the program for making effective decisions. Our *abstract conflict analysis* learns abstract transformers over a given template following a UIP computation. Experimental evaluation over a range of benchmarks shows a 20x reduction in the total number of *decisions*, *propagations*, *conflicts* and *backtracking* iterations compared to CBMC. Moreover, ACDLP is 1.5x faster than CBMC. Compared to Astrée, ACDLP solves twice as many benchmarks and has much higher precision. In the future, we plan to extend our framework to unbounded verification through invariant generation.

References

1. Astrée, `https://www.absint.com/astree/index.htm`

2. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
3. Biere, A., Heule, M., Van Maaren, H., Walsh, T.: Handbook of Satisfiability. IOS (2009)
4. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Programming Language Design and Implementation (PLDI). pp. 196–207. ACM (2003)
5. Brain, M., D'Silva, V., Griggio, A., Haller, L., Kroening, D.: Deciding floating-point logic with abstract conflict driven clause learning. Formal Methods in System Design 45(2), 213–245 (2014)
6. Brain, M., Joshi, S., Kroening, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: Static Analysis Symposium (SAS). LNCS, vol. 9291, pp. 145–161. Springer (2015)
7. CBMC, `http://www.cprover.org/cbmc/`
8. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. pp. 168–176. LNCS, Springer (2004)
9. Cotton, S.: Natural domain SMT: A preliminary assessment. In: Formal Modeling and Analysis of Timed Systems (FORMATS). LNCS, vol. 6246, pp. 77–91. Springer (2010)
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages (POPL). pp. 238–252. ACM (1977)
11. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Principles of Programming Languages (POPL). pp. 269–282. ACM (1979)
12. CPROVER verification framework. `http://www.cprover.org/`
13. D'Silva, V., Haller, L., Kroening, D.: Abstract conflict driven learning. In: Principles of Programming Languages (POPL). pp. 143–154. ACM (2013)
14. D'Silva, V., Haller, L., Kroening, D., Tautschnig, M.: Numeric bounds analysis with conflict-driven learning. In: TACAS. pp. 48–63. LNCS, Springer (2012)
15. D'Silva, V., Kroening, D., Haller, L.: Satisfiability solvers are static analysers. In: Static Analysis Symposium (SAS). pp. 317–333. LNCS, Springer (2012)
16. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. Journal on Satisfiability, Boolean Modeling and Computation 1, 209–236 (2007)
17. Haller, L.C.R.: Abstract satisfaction. Ph.D. thesis, University of Oxford, UK (2013)
18. Harris, W.R., Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program analysis via satisfiability modulo path programs. In: Principles of Programming Languages (POPL). pp. 71–82. ACM (2010)
19. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV. pp. 661–667. LNCS, Springer (2009)
20. McMillan, K.L., Kühlmann, A., Sagiv, M.: Generalizing DPLL to richer logics. In: CAV. pp. 462–476. LNCS, Springer (2009)
21. Minisat, `http://minisat.se/`
22. de Moura, L.M., Jovanovic, D.: A model-constructing satisfiability calculus. In: VMCAI. pp. 1–12. LNCS, Springer (2013)
23. Mukherjee, R., Schrammel, P., Haller, L., Kroening, D., Melham, T.: Lifting CDCL to template-based abstract domains for program verification (extended version). arXiv Computing Research Repository arXiv:1707.02011 [cs.LO] (July 2017)
24. Mukherjee, R., Tautschnig, M., Kroening, D.: v2c – A verilog to C translator. In: TACAS. pp. 580–586. LNCS, Springer (2016)
25. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM TOPLAS 29(5) (2007)
26. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: VMCAI. pp. 25–41. LNCS, Springer (2005)

27. Schrammel, P., Kroening, D.: 2LS for program analysis (competition contribution). In: TACAS. pp. 905–907. LNCS, Springer (2016)