Numeric Bounds Analysis with Conflict-Driven Learning*

Vijay D'Silva**, Leopold Haller, Daniel Kroening, and Michael Tautschnig

Computer Science Department, University of Oxford
name.surname@cs.ox.ac.uk

Abstract. This paper presents a sound and complete analysis for determining the range of floating-point variables in control software. Existing approaches to bounds analysis either use convex abstract domains and are efficient but imprecise, or use floating-point decision procedures, and are precise but do not scale. We present a new analysis that elevates the architecture of a modern SAT solver to operate over floating-point intervals. In experiments, our analyser is consistently more precise than a state-of-the-art static analyser and significantly outperforms floating-point decision procedures.

1 Introduction

Automotive and avionic control software has a special structure. Few programming language constructs are used, pointers are avoided and loop iterations are often bounded by constants. Nonetheless, such software performs complex tasks, computing vehicle trajectories and approximating non-linear functions. Control software verification involves proving that IEEE 754 floating-point operations in programs are free of overflows and approximation errors. We present a new, sound and complete analysis for this problem, and demonstrate empirically that the analysis is more efficient and precise than the state of the art.

Bounds checking is the problem of determining if the value of a numeric variable lies in a given range. *Interval analysis*, a classic approach to bounds checking, propagates intervals through a program. Intervals analysis is extremely fast but woefully imprecise, producing proofs on only 17 of 33 of our safe benchmarks (see § 5). Another shortcoming of interval analysis is that imprecision cannot be distinguished from errors. An alternative approach to bounds checking is bounded model checking (BMC) with an IEEE 754 decision procedure. BMC is precise but does not scale: of 57 benchmarks, only 23 can be solved by BMC within a minute, whereas interval analysis usually requires less than a second. Another problem is that unbounded loops cannot be handled directly.

We present *Conflict Driven Fixed Point Learning* (CDFL), a new program analysis that embeds an abstract domain inside the Conflict Driven Clause

* Supported by the Toyota Motor Corporation, EPSRC project EP/H017585/1 and ERC project 280053.

** Supported by a Microsoft Research European PhD Scholarship.

Learning (CDCL) algorithm of modern SAT solvers. A SAT solver uses constraint propagation, decisions, backtracking, a conflict graph, and clause learning to decide satisfiability. We develop abstract domain analogues of these ideas: Constraint propagation uses fixed point iteration, decisions restrict the range of intervals, the conflict graph is labelled with intervals, and learning generates program analysis constraints in place of propositional clauses.

CDFL is both a static analyser and a decision procedure. From a static analysis perspective, CDFL is an abstract interpreter that uses decisions and learning to increase transformer precision. From a decision procedure perspective, CDFL is a SAT solver for program analysis constraints. CDFL is a strict generalisation of propositional CDCL in that, on acyclic programs with only Boolean variables, our analyser is a clause-learning SAT solver. Elucidating this connection is beyond the scope of this paper.

Contribution and Contents Our new interval analysis builds on the following contributions to combine the strength of static analysis and BMC.

- A novel account of program safety as satisfiability of a set-constraint formula. Unlike the standard formulation of static analysis, which focuses on invariants, our formulation is based on error traces.
- A new interval analysis that exploits the efficiency of the interval domain while being path-sensitive and bit-level accurate.
- A tool that can prove correctness of non-linear, IEEE 754 floating-point computations using *only the interval abstraction*. Our experiments reveal that existing techniques are either imprecise or slow on such programs.

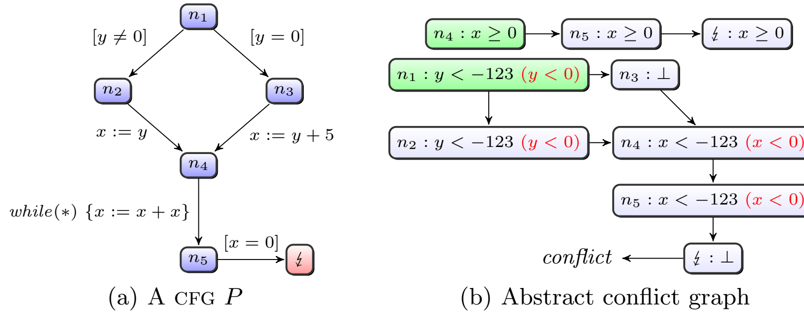
The rest of this section illustrates our approach and discusses related work. The new formulation of safety as satisfiability is in § 3 and our procedure for deciding satisfiability is in § 4. Implementation and benchmarks are discussed in § 5.

1.1 Overview

A program, as in Figure 1(a), is an acyclic control flow graph (ACFG) Edges can be labelled with loops, so this representation is not limiting. The variables x and y are mathematical integers, $[y = 0]$ is a test, and $*$ denotes non-deterministic choice. We wish to determine if the error location $\lightning$ is reachable.

The analysis associates an interval with each location and variable. The intervals for x and y at n_1 are $[-\infty, \infty]$. The condition $[y \neq 0]$ cannot be modelled by an interval, so the interval for y at n_2 is $[-\infty, \infty]$. The intervals for x at n_4 and n_5 are $[-\infty, \infty]$, so the analysis cannot prove safety.

The analysis is refined using (for now, arbitrary) constraints on intervals. First, x is constrained to be in $[0, \infty]$ at n_4. Interval analysis concludes that x is in $[0, \infty]$ at n_5 but cannot prove safety. A second decision constrains y to $[-\infty, -124]$ at n_1. Interval analysis shows x to be in $[-\infty, -124]$ at n_5, so P is safe assuming x is in $[0, \infty]$ at n_4 and y is in $[-\infty, -124]$ at n_1. A proof by cases



(a) A CFG P (b) Abstract conflict graph

Fig. 1. A control flow graph and conflict graph.

would repeat the analysis, once with x in $[-\infty, -1]$ at n_4 and once with y in $[-123, \infty]$ at n_1. *We do not do a proof by cases.*

Learning is used to avoid enumerating cases. Deductions made during fixed point iteration are represented by the abstract conflict graph in Figure 1(b). For instance, the fact $x \geq 0$ at n_4 implies that the same holds at n_5 and $\lightning$. The decisions are $(n_4 : x \geq 0)$ and $(n_1 : y < -123)$, but only the latter is connected to $(\lightning : \bot)$, and suffices to prove safety. Conflict analysis in SAT solvers is similar. The next step is new and does not exist in SAT solvers. Constraints in the graph are generalised to the labels in parentheses. In Figure 1(b), $(n_1 : y < -123)$ generalises to $(n_1 : y < 0)$; we *learn* that all error traces must satisfy $y \geq 0$ at n_1. Our analysis backtracks, discarding all assumptions. Interval analysis is run with the learnt constraint and can prove safety.

We emphasise that learning and case-based reasoning are different. A case-based method does a proof under each assumption. Learning, however, generates constraints that preserve error reachability. These constraints are not assumptions. The procedure is simply rerun after learning.

1.2 Related Work

CDFL can be understood from the perspective of both static analysis and SAT solving. As a static analysis, CDFL is an automated refinement technique. Possibly the best-known refinement technique is CEGAR [5], which uses spurious counterexamples to refine an abstract domain and synthesise transformers. CDFL restricts transformers to eliminate sources of imprecision. The expensive operations of domain refinement and transformer synthesis are avoided.

Other work that refines an analysis without modifying the domain eliminates joins [11], constrains widening operators [23, 22] or transformers [10]. In [10], a counterexample DAG is analysed and interpolants are used to constrain the analysis. Our work is similar, but a conflict graph is analysed and decisions and clauses are used to constrain the analysis. All these techniques can be viewed as instances of *trace partitioning* [19]. In technical terms, CDFL discovers program-

and property-specific trace partitions. Other methods to discover trace partitions use CFG rewriting [20], and regular languages [2, 12, 20].

Let us discuss decision procedures. The DPLL(T) architecture leverages the efficiency of SAT solvers to reason about richer theories but separates propositional and theory reasoning. An alternative is to lift CDCL directly to a first-order theory as in generalised DPLL [17], conflict resolution [15], natural-domain SMT [7], and the cutting-planes extension [14]. Our work has similar motivations to these. First order formulae are replaced by abstract domain elements, transformers do theory propagation and set constraints are learnt instead of lemmas. We believe the abstract conflict graph and the related generalisation are new. Moreover, set-constraints allow us to handle loops.

Finally, CDFL is one of many efforts to combine static analysis and decision procedures. *Shallow integration* methods facilitate communication between engines treated as black boxes. Satisfiability Modulo Path Programs (SMPP) [13] lifts DPLL(T) to programs, using a SAT solver to guide an abstract interpreter. *Deep integration* techniques embed program analysis in satisfiability architectures. Examples are YOGI [3] and lazy annotation [16], both of which require quantifier-free first order theories. We use abstract domains and transformers.

2 Program Model and Domain

This section introduces the program model and identifies properties of the interval domain that enable conflict driven learning.

Programs Consider sets of expressions Exp and Boolean expressions BExp over variables Var. We focus on numeric, machine data types, so variables take values in a finite, totally ordered set Val with minimum min and maximum max. IEEE 754 values are ordered by setting NaN greater than numeric values and using the arithmetic order. A statement, as below, is an assignment, conditional, sequential concatenation, non-deterministic choice or a loop.

$$s ::= x := \mathit{exp} \;\mid\; [b] \;\mid\; s_1; s_2 \;\mid\; \mathit{choose}\{s_1, s_2\} \;\mid\; \mathit{loop}\{s\}$$

An *acyclic control flow graph* (ACFG) is an acyclic graph $(\mathit{Loc}, E, \mathit{stmt})$, with locations Loc, edges E and a function stmt labelling edges with statements. Loc contains unique initial and error locations init and $\lightning$. A *loop-free* ACFG has no *loop* statements.

Concrete Semantics Statement semantics is defined over environments $\mathit{Env} = \mathit{Var} \to \mathit{Val}$. A statement s defines a function $\mathit{post}_s : \wp(\mathit{Env}) \to \wp(\mathit{Env})$, called a transformer. Assignments and tests have their expected semantics; concatenation is composition, choice is union and the semantics of a loop is a fixed point.

$$\mathit{post}_{s_1;s_2} = \mathit{post}_{s_2} \circ \mathit{post}_{s_1} \qquad \mathit{post}_{\mathit{choose}\{s_1,s_2\}} = \lambda X.\mathit{post}_{s_1}(X) \cup \mathit{post}_{s_2}(X)$$
$$\mathit{post}_{\mathit{loop}\{s\}} = \lambda X.\mathsf{lfp} Y. X \cup \mathit{post}_s(Y)$$

We abbreviate $post_{stmt(n,m)}$ to $post_{(n_i,n_{i+1})}$. A *state* is a location with an environment. A *trace* is a sequence of states $(n_0, \varepsilon_0), \ldots, (n_k, \varepsilon_k)$ such that for all $0 \leq i < k$, (n_i, n_{i+1}) is an ACFG edge and $\varepsilon_{i+1} \in post_{(n_i,n_{i+1})}(\{\varepsilon_i\})$. A program is *safe* if there is no trace as above with $n_0 = init$ and $n_k = \lightning$. The *concrete domain* is the lattice of environments $\wp(Var \to Val)$ with a transformer $post_s$ for statements. Abstract interpretation with intervals is illustrated next.

Example 1. We use abstract interpretation to show that, if x is in the range $[-3, 3]$ and the statement $s = z := x; y := x * z$ is executed, y is non-negative. Let $a = \langle x \mapsto [-3, 3]\rangle$ denote that x has the range shown and other variables are unconstrained. Abstract transformers, denoted by $\hat{post}$, produce the facts below.

$$\hat{post}_{z:=x}(a) = b = \langle x \mapsto [-3, 3], z \mapsto [-3, 3]\rangle$$
$$\hat{post}_{y:=x*z}(b) = \langle x \mapsto [-3, 3], y \mapsto [-9, 9], z \mapsto [-3, 3]\rangle$$

Intervals lose the information that x and z are equal. Let m be $\langle x \mapsto [0, max]\rangle$ and $\sim m$ be $\langle x \mapsto [min, -1]\rangle$. The two intervals cover all values of x. Precision is regained by rerunning the analysis with the restrictions below.

$$\hat{post}_s(a \sqcap m) \sqcup \hat{post}_s(a \sqcap \sim m) = \langle x \mapsto [-3, 3], y \mapsto [0, 9], z \mapsto [-3, 3]\rangle$$

The transformer restriction increases precision without losing soundness. $\lhd$

Interval Abstraction The set Itv of intervals over Val contains pairs $[l, u]$ with $l \leq u$. The partial order on Itv is: $a \sqsubseteq b$ if b contains a. The *interval environments* domain is the lattice $IEnv = ((Var \to Itv) \cup \{\bot\}, \sqsubseteq, \sqcup, \sqcap)$ with abstract transformers $\hat{post}_s$. The least element is $\bot$ and the greatest element $\top$ maps all variables to $[min, max]$. The interval environment that maps x_1 to $[l_1, u_1]$, x_2 to $[l_2, u_2]$ and all other variables to $[min, max]$ is denoted $\langle x_1 \mapsto [l_1, u_1], x_2 \mapsto [l_2, u_2]\rangle$. Further, $\langle x \mapsto [min, c]\rangle$ is written $\langle x \leq c\rangle$. The well known Galois connection $\wp(Env) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} IEnv$ between environments and interval environments is recalled below.

$$\alpha(\emptyset) = \bot \qquad \alpha(X) = \left\{ x \mapsto [\inf_{\varepsilon \in X} \{\varepsilon(x)\}, \sup_{\varepsilon \in X} \{\varepsilon(x)\}] \mid x \in Var \right\}$$
$$\gamma(\bot) = \emptyset \qquad \gamma(a) = \{\varepsilon \mid \varepsilon(x) \text{ is in } a(x) \text{ for all } x\}$$

Atoms and Meet Irreducibles The properties of elements used for decisions and clause learning are identified next. Fix a lattice $(A, \sqsubseteq, \sqcup, \sqcap)$ with elements $\bot$ and $\top$. An *atom* x of A is a least element strictly above $\bot$: $\bot \sqsubset x$ and no y satisfies $\bot \sqsubset y \sqsubset x$. The set of atoms is $Atoms(A)$. An atom of $\wp(Env)$ is a singleton. An atom of $IEnv$ maps variables to singleton intervals. An abstract transformer is *precise on atoms* if the equality $post_s \circ \gamma = \gamma \circ \hat{post}_s$ holds. We assume abstract transformers for loop-free statements have this property.

An element a is *meet irreducible* if, for all $X \subseteq A$, $\bigsqcap X = a$ implies a is in X. The meet irreducibles of A are $Irred_{\sqcap}(A)$. Meet irreducibles of $\wp(Env)$

are complements of singletons, and those of *IEnv* have the form $\langle x \leq c\rangle$ or $\langle x \geq c\rangle$. A *meet decomposition function* $decomp : A \to \wp(Irred_{\sqcap}(A))$ satisfies that $\bigsqcap decomp(a) = a$ for all a.

Example 2. The element $m = \langle x \geq 0\rangle$ in Example 1 is meet irreducible. The complement of m in the concrete, $Env \setminus \gamma(m)$, has a precise representation in the abstract as $\sim m = \langle x < 0\rangle$. Interval environments lack complements but can be decomposed into meet-irreducibles that have complements.

$$decomp(\langle x \mapsto [0, max], y \mapsto [1, 4]\rangle) = \{\langle x \geq 0\rangle, \langle y \geq 1\rangle, \langle y < 3\rangle\}$$

Complementable meet irreducibles are required for CDCL. ◁

An interval environment a is *precisely complementable* if there is an abstract element $\sim a$ satisfying $\gamma(a) = \wp(Env) \setminus \gamma(\sim a)$. Precise complementation differs from the standard notion of a complement in a lattice. Interval environments have the property that all meet irreducibles are precisely complementable.

3 Static Analysis as Second-Order Constraint Solving

A standard approach to analyse programs is to solve a set of equations generated by a control-flow graph. Program analysis with set-constraints makes the language explicit [8, 1]. A constraint language and its satisfiability problem are defined next. Fix an ACFG $P = (Loc, E, stmt)$, a set $CVar = \{X_n | n \in Loc\}$ of *constraint variables* indexed by locations and a concrete domain $\wp(Env)$.

Second-Order Constraints *Terms* and *constraints* are of the form below, with d ranging over concrete domain elements, and m and n over locations.

$$\begin{array}{ll} \textit{terms} & t ::= d \mid X_n \mid post_{(m,n)}(t) \mid t \cup t \mid t \cap t \\ \textit{constraints} & c ::= X_n \subseteq t \mid X_n \supseteq t \mid X_n \cap t \supset \emptyset \end{array}$$

Let a be an interval environment. We abuse notation and write $X_l \cap a$ for $X_l \cap \gamma(a)$. A *clause* is a disjunction of constraints and a *formula* is a conjunction of clauses. Following standard convention, a clause is represented as a set of constraints and a formula as a set of clauses.

A *valuation* $v : Loc \to \wp(Env)$ maps constraint variables to sets of environments. Valuations form a lattice $(CVals, \sqsubseteq, \sqcup, \sqcap)$. The order, join and meet are lifted pointwise from $\wp(Env)$. That is, $v_1 \sqsubseteq v_2$ if $v_1(l) \sqsubseteq v_2(l)$ for all locations l, and $v_1 \oplus v_2 = \lambda l.\ v_1(l) \oplus v_2(l)$ for $\oplus$ in $\{\sqcap, \sqcup\}$. An *atomic valuation* maps every location to at most one environment. The semantics $[\![t]\!]_v$ of a term t under a valuation v is inductively defined below.

$$\begin{array}{ccc} [\![d]\!]_v = d \qquad [\![X_n]\!]_v = v(n) & & [\![t_1 \cup t_2]\!]_v = [\![t_1]\!]_v \cup [\![t_2]\!]_v \\ [\![post_{(m,n)}(t)]\!]_v = post_{(m,n)}([\![t]\!]_v) & & [\![t_1 \cap t_2]\!]_v = [\![t_1]\!]_v \cap [\![t_2]\!]_v \end{array}$$

A valuation v satisfies a constraint $t_1 \bowtie t_2$ if $[\![t_1]\!]_v \bowtie [\![t_2]\!]_v$ holds for $\bowtie$ in $\{\subseteq, \subset\}$. A valuation satisfies a clause if it satisfies at least one constraint in the clause and satisfies a formula if every clause in the formula is satisfied. A valuation satisfying a formula is a *solution*. A formula is *satisfiable* if it has a solution.

3.1 Safety as Satisfiability

The standard approach to checking program safety is to compute an invariant. Formally, an invariant is a solution to the formula $Inv(P)$ below.

$$Inv(P) = X_{init} \supseteq Env \wedge \bigwedge_{n \in Loc} \left\{ X_n \supseteq \bigcup_{(m,n)\in E} post_{(m,n)}(X_m) \right\}$$

The error is unreachable if an invariant also satisfies the formula $X_{\lightning} \subseteq \emptyset$. Standard static analysis for safety can be viewed as a sound but incomplete SAT procedure for the formula $Safe(P) = Inv(P) \wedge X_{\lightning} \subseteq \emptyset$. An alternative we propose is to search for an error – a solution to the formula below.

$$Exec(P) = X_{init} \subseteq Env \;\wedge\; \bigwedge_{n \,\in\, Loc} \left\{ \bigvee_{(m,n)\in E} X_n \subseteq post_{(m,n)}(X_m) \right\}$$

A program contains an error if a solution to $Exec(P)$ also satisfies $X_{\lightning} \supset \emptyset$. BMC can be viewed as a SAT procedure for the formula $Err(P) = Exec(P) \wedge X_{\lightning} \supset \emptyset$. Solutions to $Safe(P)$ and $Err(P)$ are quite different as demonstrated next.

Example 3. Revisit the ACFG P in Figure 1. An environment ε is written as $(\varepsilon(x), \varepsilon(y))$. The valuation v_1 that maps all locations to Env is an invariant and satisfies $Inv(P)$, as does $v_2 = \{n_1 \mapsto Env, n_2 \mapsto \{(i,j)|j \neq 0\}, n_3 \mapsto \{(i,j)|j = 0\}, n_4 \mapsto \{(i,j)|i \neq 0\}, n_5 \mapsto \{(i,j)|i \neq 0\}, \lightning \mapsto \emptyset\}$. Only v_2 satisfies $Safe(P)$ and is strong enough to prove safety.

The condition $X_{\lightning} \supset \emptyset$ prevents v_2 from satisfying $Err(P)$. The constraint $X_{\lightning} \subseteq post_{[x=0]}(X_4)$ is not satisfied by v_1 so neither is $Err(P)$. In fact, $Err(P)$ is unsatisfiable. Let P' be the ACFG with the test $[y = 0]$ modified to $[y \leq 0]$. The valuation $v_3 = \{n_1 \mapsto \{(1,-5),(3,-7)\}, n_2 \mapsto \emptyset, n_3 \mapsto \{(1,-5)\}, n_4 \mapsto \{(0,-5)\}, n_5 \mapsto \{(0,-5)\}, \lightning \mapsto \{(0,-5)\}\}$ contains states on an error trace. This valuation does not satisfy $Inv(P')$ or $Safe(P')$ but satisfies $Err(P')$. $\lhd$

To prove safety of P, we can either find an invariant satisfying $Safe(P)$ or show that $Err(P)$ is unsatisfiable.

Lemma 1. *The following conditions are equivalent for an* ACFG P. *(1)* P *is safe. (2)* $Safe(P)$ *is satisfiable. (3)* $Err(P)$ *is unsatisfiable.*

In propositional SAT solvers, a partial assignment represents a set of potential solutions to a formula. Decisions and constraint propagation refine this set. Over programs, we represent potential sets of errors and use transformer restriction and fixed point iteration to refine the set.

Abstract Valuations An abstract valuation is figuratively an envelope containing potential solutions to $Err(P)$. An *abstract valuation* maps constraint variables to interval environments. An abstract valuation is *atomic* if it maps each constraint variable to an atom or to $\bot$. The abstract semantics $\|t\|_v$ of a term t with respect to an abstract valuation v is defined as expected, with abstract transformers, join and meet replacing concrete ones. An abstract valuation $\hat{v}$ *abstractly satisfies* a formula if there is a concrete solution v for which the inequality $v(X) \subseteq \gamma \circ \hat{v}(X)$ holds for all constraint variables. If the formula $Err(P)$ cannot be abstractly satisfied, the program is safe.

Example 4. Consider the ACFG $init \xrightarrow{[x<0]} n \xrightarrow{[x=4]} \lightning$ generating the formula below.

$$Err(P) = X_{init} \subseteq Env \wedge X_n \subseteq post_{[x<0]}(X_{init}) \wedge X_{\lightning} \subseteq post_{[x=4]}(X_n)$$

Standard static analysis can be viewed as refining an abstract valuation as below.

$$\begin{aligned}
\hat{v}_0 &= \{X_{init} \mapsto \top, X_n \mapsto \top, X_{\lightning} \mapsto \top\} \\
\hat{v}_1 &= \{X_{init} \mapsto \top, X_n \mapsto \langle x < 0 \rangle, X_{\lightning} \mapsto \top\} \\
\hat{v}_2 &= \{X_{init} \mapsto \top, X_n \mapsto \langle x < 0 \rangle, X_{\lightning} \mapsto \bot\}
\end{aligned}$$

As $X_{\lightning}$ maps to $\bot$, $Err(P)$ is not abstractly satisfied, so $\lightning$ is unreachable. $\triangleleft$

4 Conflict Driven Fixed Point Learning

We now present CDFL, a procedure that lifts propositional CDCL to abstract domains and program analysis constraints. Example 4 showed that standard static analysis can be viewed as a process that applies transformers to refine an abstract valuation. CDFL extends standard static analysis by using decisions, deduction, learning and backtracking to search the space of abstract valuations. Decisions restrict abstract domain elements, deduction uses transformers and set-constraint clauses and learning infers set-constraint clauses. For simplicity, heuristics for learning, decision making, and backtracking are abstracted away as non-deterministic choices. Common heuristics described in the SAT literature such as first-UIP learning, non-chronological backtracking, restarts and activity-based decision heuristics can be used to resolve this non-determinism.

4.1 Overview of CDFL

The technique is shown in Procedure CDFL. It begins with a formula $Err(P)$ and the abstract valuation $v = \lambda X.\top$. The call deduce() refines this valuation to the result of standard fixed point iteration in an abstract domain. If static analysis shows that the program is safe, our procedure terminates. Otherwise, static analysis was not precise enough and the solver enters the main loop. Thus, CDFL never does extra work if standard static analysis is sufficiently precise.

The current valuation is refined using an interval meet irreducible in the call to decide(). Consequences of this decision are inferred by a call to deduce().

```
1  (v, F) ← (λl.⊤, Err(P))
2  deduce()
3  if v(↯) = ⊥ then return safe
4  while true do
5  |  if atomic(v) then return (v, fail)
6  |  decide()
7  |  deduce()
8  |  while (v, F) in conflict do
9  |  |  learn()
10 |  |  if backtrack() = fail then return safe
11 |  |  deduce(v, F)
```

Procedure CDFL: Conflict Driven Fixed Point Learning

Decisions and deduction alternate until one of two scenarios. If $\mathsf{atomic}(v)$ returns true, the valuation v cannot be refined and is returned. Either v contains an error trace or the current abstraction is insufficient to prove safety. The second scenario is a conflict; the valuation v does not abstractly satisfy the formula. Learning is used to generate a reason for the conflict. Technically, learning adds a clause C to the current formula, so that $F \wedge C$ is equi-satisfiable to F. The backtracking step $\mathsf{backtrack}()$ then returns the solver to an earlier state that does not conflict with C. If this is not possible, $Err(P)$ is unsatisfiable and P is safe.

4.2 Data Structure and Phases of CDFL

Internally, SAT solvers use a stack to track the sequence of variable assignments of the form (x, v) where x is a propositional variable, and v is either true or false. In our procedure, the stack contains elements of the form (l, a), where l is a location and a is a meet-irreducible or is $\bot$.

A *labelled restriction* (l, a, z) consists of a location l, a meet-irreducible a and the label $z = \mathsf{d}$ if (l, a) is a decision, or $z = \mathsf{i}$ if (l, a) was inferred by deduction. The set of labelled restrictions is $\mathcal{L} = Loc \times (Irred_{\sqcap}(IEnv) \cup \{\bot\}) \times \{\mathsf{d}, \mathsf{i}\}$. A *stack* is a sequence of labelled restrictions, where the empty stack is ϵ, and UV denotes concatenation. A stack S defines an abstract valuation $\lfloor S \rfloor$ where $\lfloor \epsilon \rfloor = \top$ and $\lfloor S(l, a, z) \rfloor = \lfloor S \rfloor \sqcap \langle l \mapsto a \rangle$. An interval meet-irreducible a at location l *refines the stack*, denoted $\mathsf{refines}(S, (l, a))$, if the condition $\lfloor S \rfloor(X_l) \sqcap a \sqsubset \lfloor S \rfloor(X_l)$ holds.

A *solver state* (S, F) consists of a stack of labelled restrictions S and a formula F and the *current valuation* is $\lfloor S \rfloor$. The solver is *in conflict* if some clause in F is not abstractly satisfied by $\lfloor S \rfloor$. We present the components of CDFL as state transitions made by the solver, inspired by the presentation of CDCL in [18].

Deduction Deduction uses two rules to transform the solver state. The rule tprop applies transformers to abstract valuations, and is comparable to theory propagation in SMT solvers. The rule uprop generalises the unit rule in proposi-

tional solvers to set-constraint clauses. If deduction refines the current valuation, the new information is added to the stack. These rules are illustrated below.

Example 5. Consider the formula $Err(P)$ for the ACFG in Figure 1. The initial valuation is $v_0 = \lambda X.\top$. This valuation is refined using the clause $\{X_3 \subseteq post_{[y=0]}(X_1)\}$, and the transformer $\hat{post}_{[y=0]}$ to v_1 that maps X_3 to $\langle y = 0\rangle$. Next, consider the clause $\{X_4 \subseteq post_{[x:=y+5]}(X_3), X_4 \subseteq post_{[x:=y]}(X_2)\}$. The right side of the first constraint evaluates to $a_1 = \langle y \mapsto [0,0], x \mapsto [5,5]\rangle$, and the second to $a_2 = \top$. One of these constraints must hold, so the weaker condition $X_4 \subseteq \gamma(a_1) \cup \gamma(a_2)$ must as well. But $a_1 \sqcup a_2 = \top$, which does not refine the current valuation of X_4, so the stack is not modified.

To illustrate the unit rule, continue with the valuation obtained above and assume that there is a clause $\{X_1 \cap \langle y < 0\rangle \supset \emptyset, X_4 \cap \langle x > 10\rangle \supset \emptyset\}$. The valuation v_1 does not satisfy the first constraint, so every solution must satisfy the second constraint. Every solution satisfying this constraint must also satisfy $X_4 \subseteq \langle x > 10\rangle$, so the valuation v_1 is refined, mapping X_4 to $\langle x > 10\rangle$. $\lhd$

The deduction rules are defined below.

$$\mathsf{tprop}: \quad (S,F) \to (S(l,a,\mathsf{i}),F) \quad \text{if } \mathsf{refines}(S,(l,a)),\ \{X_l \subseteq t_1,\ldots,X_l \subseteq t_k\} \in F \text{ where } a \in decomp\Big(\bigsqcup_{1\le i\le k} \|t_i\|_{\lfloor S\rfloor} \sqcap \lfloor S\rfloor(X_l)\Big)$$

$$\mathsf{uprop}: \quad (S,F) \to (S(l,a,\mathsf{i}),F) \quad \text{if } \mathsf{refines}(S,(l,a)) \text{ and } (\{X_l \cap t \supset \emptyset\} \cup C) \in F \text{ where } C \text{ is not abstractly satisfied by } \lfloor S\rfloor \text{ and where } a \in decomp(\|t\|_{\lfloor S\rfloor})$$

Both of these rules are sound in the sense that if $\lfloor S\rfloor$ contains a solution to F, $\lfloor S(l,a,\mathsf{i})\rfloor$ will also contain a solution. The function deduce applies these rules exhaustively until the valuation becomes atomic or until the solver is in conflict.

Decisions A decision picks a location l and program variable x and constrains it with a meet irreducible. Additionally, decisions must be chosen such that they do not put the solver in conflict.

$$\mathsf{decide}: \quad (S,F) \longrightarrow (S(l,a,\mathsf{d}),F) \quad \text{if } \mathsf{refines}(S,(l,a)) \text{ and } (S(l,a,\mathsf{d}),F) \text{ is not in conflict}$$

Example 6. A valid decision for the CFG P in Figure 1, for the valuation v that maps $X_{\lightning}$ to $\langle x \mapsto [0,0]\rangle$ and all other locations to $\top$ is $(X_1, \langle y > 0\rangle, \mathsf{d})$. The restriction $(\lightning, \langle x > 0\rangle, \mathsf{d})$ is *not* a valid decision because $v(X_{\lightning}) \sqcap \langle x > 0\rangle$ is bottom, causing a conflict. $\lhd$

Learning and Backtracking Learning identifies sufficient reasons for conflicts and adds a clause that expresses the negation of that reason. For an abstract valuation v, we define the *clause complement* $clcomp(v)$ as the clause

$\{X_l \cap {\sim}a \supset \emptyset \mid a \in \mathit{decomp}(v(l))\}$. This formula is a complement in the sense that a concrete valuation is a solution of $\mathit{clcomp}(v)$ exactly if it is not contained in the concretisation of v.

Example 7. Let $\mathit{Loc} = \{1, 2\}$ and let v be an abstract valuation with $v(1) = \langle x < 0\rangle$ and $v(2) = \langle y \mapsto [0, 10]\rangle$. Then $\mathit{clcomp}(v)$ contains the three constraints $X_1 \cap \langle x \geq 0\rangle \supset \emptyset$, $X_2 \cap \langle y < 0\rangle \supset \emptyset$ and $X_2 \cap \langle y > 10\rangle \supset \emptyset\}$. $\lhd$

Backtracking is used to remove a suffix of the stack to restore the solver to a consistent state after a conflict has been encountered. Backtracking may only jump back to decision elements on the stack. Abstract rules for learning and backtracking can be stated as follows.

$$\begin{array}{llll} \mathsf{learn}: & (S, F) \rightarrow (S, F \wedge \mathit{clcomp}(R)) & \text{if } \mathit{clcomp}(R) \notin F \text{ and } \gamma_V(R) \\ & & \text{contains no solutions of } F \\ \mathsf{backtrack}: & (S_1(l, a, d)S_2, F) \rightarrow (S_1, F) & \text{if } (S_1, F) \text{ is not in conflict} \end{array}$$

Soundness and Completeness We denote the CDFL procedure over the interval domain as CDFL(IEnv). The CDFL(IEnv) procedure is sound, and, under certain conditions, also complete.

Theorem 1. *If P is a loop-free program and the set of values Val is finite, then* CDFL(IEnv) *is a sound and complete decision procedure to check safety of P.*

4.3 Abstract Conflict Graphs

In order to instantiate the learning step, heuristics for finding conflict reasons are needed. Like propositional solvers, we record deductions in a data structure called *conflict graph*, which is incrementally built by recording decisions and deductions. The nodes of the conflict graph are labelled restrictions. An example is provided in Figure 1. The two nodes without predecessors are decision nodes, all other nodes are implication nodes. The predecessors of each node n in the graph are sufficient to deduce n. Once a conflict is reached, the graph is analysed to determine a sufficient reason for unsatisfiability. A *cut* of a conflict graph (R, I) is a set $L \subseteq R$ such that any path from a decision node to the conflict node goes through L. Every cut of a conflict graph provides a conflict reason that can be used in learning. In contrast to propositional SAT, we can obtain stronger learnt clauses by generalising the nodes of the implication graph itself before obtaining a cut. Generalisation is performed by computing maximal sufficient pre-conditions in the domain of intervals, which we handle in our implementation using binary search on bounds.

Example 8. Consider the conflict graph in Figure 1. The following two sets are different cuts of the graph, and hence sufficient reasons for a conflict, $R_1 = \{n_5 : x < -123\}$, $R_2 = \{n_2 : y < -123, n_3 : \bot\}$. In learning, these cuts produce the following clauses, $C_1 = \{X_5 \cap \langle x \geq -123\rangle \supset \emptyset\}$, $C_2 = \{X_2 \cap \langle y \geq -123\rangle \supset \emptyset, X_3 \supset \emptyset\}$. Stronger clauses can be obtained by applying generalisation first.

Consider the node $n_5 : x < -123$ in the conflict graph of Figure 1. This node is used to deduce $\lightning : \bot$ using the conditional statement $s = [x = 0]$. The weakest pre-condition of $\lightning : \bot$ w.r.t. s is $x < 0 \lor x > 0$, but this is not expressible as an interval element. Instead, we choose the maximal generalisation of $x < -123$ that is sufficient to prove $\lightning : \bot$, and obtain $x < 0$. Cutting now yields the stronger clause $\{X_5 \cap \langle x \geq 0\rangle \supset \emptyset\}$. $\triangleleft$

5 Implementation and Experiments

We have implemented CDFL for ANSI-C programs. The domains used are intervals over IEEE 754 floating-point numbers and machine integers. This section will show that our approach is able to efficiently prove correctness of several programs where a standard interval analysis yields a false alarm. In case our procedure fails to prove correctness it returns a concrete environment at the initial control flow node to constants. This assignment either leads to an error, or helps localise the imprecision of the abstract analysis by providing a maximal restriction that cannot be proved correct using intervals. We apply our analysis to verify properties on floating-point programs from various sources, and show that, in many cases, our analysis is as efficient as static analysis, but provides the precision of a floating-point decision procedure.

We compare our tool to the static analyser Astrée [4], which uses interval analysis, and to the bounded model checker CBMC [6], which uses a bit-precise floating-point decision procedure based on propositional encoding. Our benchmarks show highly non-linear behaviour. Astrée is not optimised for the kinds of programs we consider and introduces a high degree of imprecision. (Astrée offers simple trace partitioning heuristics for Booleans and machine integers, but not floating-point programs.) CBMC translates the floating-point arithmetic to large propositional circuits which are hard for SAT solvers. As benchmarks we use ANSI-C code originating from (a) controller code auto-generated from a Simulink model with varying loop bounds; (b) examples from the Functional Equivalence Verification Suite [21]; (c) benchmarks presented at the 2010 Workshop on Numerical Software Verification; (d) code presented by Goubault and Putot [9]; (e) hand-crafted instances that implement Taylor expansions of sine and square functions, as well as Newton-Raphson approximation. In order to allow comparison to bounded model checking, only benchmark programs with bounded loops were chosen, which were completely unrolled prior to analysis. All our 57 benchmarks, more detailed benchmark results, together with the prototype tool, are available online[1].

We discuss the following results: (1) our analysis is as precise as a full floating-point decision procedure while still being orders of magnitudes faster; (2) learning and the choice of decision heuristic yield a speed-up of more than an order of magnitude; (3) dynamic precision adjustment is observed frequently.

[1] `http://www.cprover.org/cdfpl/`

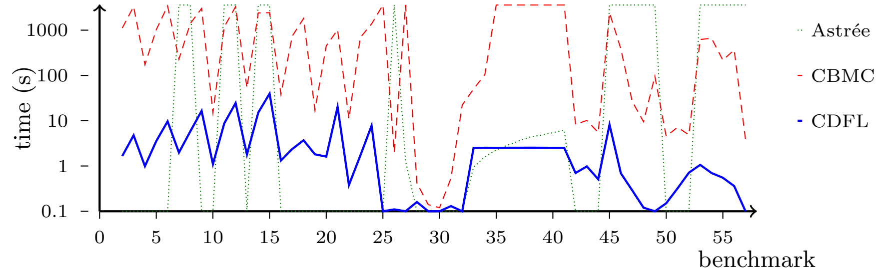


Fig. 2. Execution times of Astrée, CBMC, and CDFL; wrong results set to 3600s

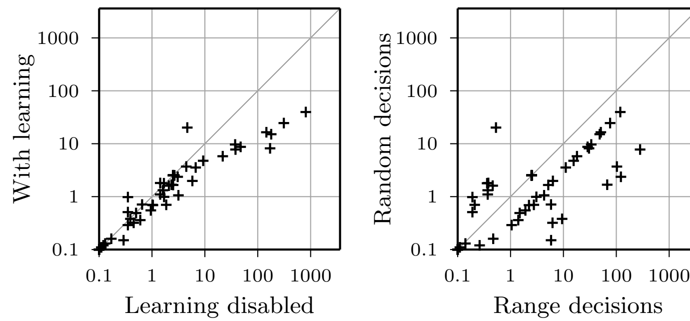


Fig. 3. Effects of learning and decision heuristics

Efficient and Precise Analysis In Figure 2, we show execution times for Astrée, CBMC, and our analysis (CDFL). To highlight wrong verification results or out-of-memory errors, the time for such failures was set to the timeout of 3600 seconds. We make several observations: on average, our analysis is at least 264 times faster than CBMC. The figure 264 is a lower bound, since some runs of CBMC were aborted due to timeouts or errors. The maximum speed-up is a factor of 1595. Although Astrée is often faster than our prototype, its precision is insufficient in many cases – we obtained 16 false alerts for the 33 safe benchmarks.

Decision Heuristics and Learning Figure 3 visualises the effects of learning and decision heuristics. Learning has a significant influence on runtime, as does the choice of a decision heuristic. We compare a random heuristic, which picks a restriction over a random variable, with a range-based one, which always aims to restrict the least restricted variable. Random decision making outperforms range-based. Activity-based heuristics common in SAT may work as well in our case.

Dynamic Precision Adjustment One unique feature of our procedure is property-dependent refinement. The precision of the analysis dynamically adapts to match the precision required by the property. This is illustrated in Figure 4

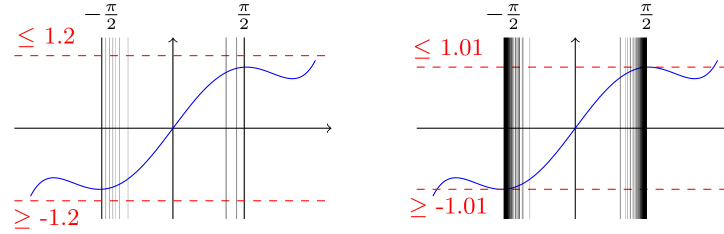


Fig. 4. Partitions explored during bounds check

where we check bounds on the result of computing a sine approximation under the input range $[-\frac{\pi}{2}, \frac{\pi}{2}]$. The input value is shown on the x-axis, the result of the computation on the y-axis. The bound we check against is depicted as two dashed horizontal lines, boundaries of explored partitions are shown as black vertical lines. The actual maximum of the function lies at about 1.00921. As the checked bound (Figure 4 shows bounds 1.2 and 1.01) approaches this value, our procedure dynamically increases the precision of the analysis by exploring more partitions.

Limitations of CDFL(*IEnv*) Our procedure is instantiated over the domain of intervals. There are simple programs that are not amenable to interval analysis, even when additional partition-based refinement is used. Consider for example the one-line program $x := y$ together with the relational property $x = y$. Intervals are non-relational, hence CDFL(*IEnv*) would enumerate all singleton intervals over y. Similar enumeration behaviour can be found in propositional SAT solvers, which may perform badly when applied to certain, highly relational problems. This can be fixed by instantiating CDCL(A) using a richer base domain. Further, our implementation is a prototype and restricts learning to the initial control flow node, which limits performance on deep programs.

6 Conclusion

We presented a novel approach for bounds analysis that instantiates a CDCL architecture over abstract domains. In the absence of loops and for finite value domains we obtain a sound and complete analysis. Our prototype implementation witnesses the potential of this approach: our analysis is substantially more precise than a state-of-the-art static analyser and outperforms a SAT based IEEE 754 floating-point decision procedure by several orders of magnitude on small, non-linear programs.

Much research in program analysis attempts to leverage the efficiency of SAT solvers. The efficiency is the result of an intensive, community effort to discover efficient engineering techniques, and decision and learning heuristics in modern solvers. This paper has demonstrated how to lift the architecture of a modern rsat solver to program analyzers. Similarly, our approach could benefit greatly by studying heuristics and efficient engineering techniques.

The formal framework in this paper is by no means limited to bounds analysis with intervals. A number of domains, numeric or otherwise, have the complementation properties necessary for instantiations of CDFL. Examples are given by numeric domains such as octagons and polyhedra, or by equality domains. Non-numeric examples include some pointer analyses, or trace-based abstractions, for example those based on control-flow unwindings, where decisions would correspond to refinements of control structure imprecision. The instantiation of CDFL with other domains is the focus of our current work. We believe that extending our technique to new domains can yield a new class of general purpose verification tools that dynamically combine the efficiency provided by abstraction with the precision of a SAT solver.

Acknowledgments We thank Antoine Miné for providing experimental results using Astrée. One anonymous reviewer summarised our paper better than we did and another pointed out technical loose ends.

References

1. A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35:79–111, November 1999.
2. G. Balakrishnan, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Refining the control structure of loops using static analysis. In *Proc. of the Intl. Conf. on Embedded Software*, pages 49–58. ACM Press, 2009.
3. N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *Proc. of Software Testing and Analysis*, pages 3–14. ACM Press, 2008.
4. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation*, volume 2566 of *LNCS*, pages 85–108. Springer, 2002.
5. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.
6. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.
7. S. Cotton. Natural domain SMT: A preliminary assessment. In *Formal Modeling and Analysis of Timed Systems*, volume 6246 of *LNCS*, pages 77–91, 2010.
8. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proc. of Functional Programming Languages and Computer Architecture*, pages 170–181. ACM Press, June 1995.
9. E. Goubault and S. Putot. Static analysis of numerical algorithms. In *Proc. of Static Analysis Symposium*, volume 4134 of *LNCS*, pages 18–34. Springer, 2006.
10. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 443–458, 2008.
11. B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, pages 474–488. Springer, 2006.

12. S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *Proc. of Programming Language Design and Implementation*, pages 375–385. ACM Press, June 2009.
13. W. R. Harris, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Program analysis via satisfiability modulo path programs. In *Proc. of Principles of Programming Languages*, pages 71–82. ACM Press, 2010.
14. D. Jovanovic and L. M. de Moura. Cutting to the chase solving linear integer arithmetic. In *Computer Aided Deduction*, volume 6803 of *LNCS*, pages 338–353. Springer, 2011.
15. K. Korovin, N. Tsiskaridze, and A. Voronkov. Conflict resolution. In *Constraint Programming*, pages 509–523, 2009.
16. K. L. McMillan. Lazy annotation for program testing and verification. In *Proc. of Computer Aided Verification*, volume 6174 of *LNCS*, pages 104–118. Springer, 2010.
17. K. L. McMillan, A. Kuehlmann, and M. Sagiv. Generalizing DPLL to richer logics. In *Proc. of Computer Aided Verification*, volume 5643 of *LNCS*, pages 462–476. Springer, 2009.
18. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.
19. X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems*, 29(5):26, 2007.
20. S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *Proc. of Static Analysis Symposium*, volume 4134, pages 3–17. Springer, 2006.
21. S. F. Siegel and T. K. Zirkel. A functional equivalence verification suite for high-performance scientific computing. Technical Report UDEL-CIS-2011/02, Department of Computer and Information Sciences, University of Delaware, 2011.
22. A. Simon and A. King. Widening polyhedra with landmarks. In *Proc. of the Asian Symposium on Programming Languages and Systems*, pages 166–182, 2006.
23. C. Wang, Z. Yang, A. Gupta, and F. Ivančic. Using counterexamples for improving the precision of reachability computation with polyhedra. In *Proc. of Computer Aided Verification*, pages 352–365. Springer, 2007.