Succinct Representations for Abstract Interpretation

Combined analysis algorithms and experimental evaluation

Julien Henry[1,2], David Monniaux[1,3], and Matthieu Moy[1,4]

[1] VERIMAG laboratory, Grenoble, France
[2] Université Joseph Fourier
[3] CNRS
[4] Grenoble-INP

Abstract. Abstract interpretation techniques can be made more precise by distinguishing paths inside loops, at the expense of possibly exponential complexity. SMT-solving techniques and sparse representations of paths and sets of paths avoid this pitfall.
We improve previously proposed techniques for guided static analysis and the generation of disjunctive invariants by combining them with techniques for succinct representations of paths and symbolic representations for transitions based on static single assignment.
Because of the non-monotonicity of the results of abstract interpretation with widening operators, it is difficult to conclude that some abstraction is more precise than another based on theoretical local precision results. We thus conducted extensive comparisons between our new techniques and previous ones, on a variety of open-source packages. We also took the opportunity to compare the precision of various classical abstract domains (convex polyhedra, octagons, intervals).

1 Introduction

Static analysis by abstract interpretation is a fully automatic program analysis method. When applied to imperative programs, it computes an inductive invariant mapping each program location (or a subset thereof) to a set of states represented symbolically [9]. For instance, if we are only interested in scalar numerical program variables, such a set may be a convex polyhedron (the set of solutions of a system of linear inequalities) [11,17,2,4].

In such an analysis, information may flow forward (one computes the polyhedron after a program statement as the image by the semantics of the statement, or a super-set thereof, of the polyhedron before) or backward; forward program analysis computes super-sets of the states reachable from the initialization of the program, backward program analysis computes super-sets of the states coreachable from some property of interest (for instance, the violation of an assertion). In forward analysis, control-flow joins correspond to convex hulls if using convex polyhedra (more generally, they correspond to least upper bounds in a

lattice); in backward analysis, it is control-flow splits that correspond to convex hulls.

It is a known limitation of program analysis by abstract interpretation that this convex hull, or more generally, least upper bound operation, may introduce states that cannot occur in the real program: for instance, the convex hull of the intervals $[-2,-1]$ and $[1,2]$ is $[-2,2]$, strictly larger than the union of the two. Such introduction may prevent proving desired program properties, for instance $\neq 0$. The alternative is to keep the union symbolic (e.g. compute using $[-2,-1] \cup [1,2]$) and thus compute in the *disjunctive completion* of the lattice, but the number of terms in the union may grow exponentially with the number of successive tests in the program to analyze, not to mention difficulties for designing suitable widening operators for enforcing the convergence of fixpoint iterations [2,4,3]. The exponential growth of the number of terms in the union may be controlled by heuristics that judiciously apply least upper bound operations, as in the *trace partitioning domain* [28] implemented in the Astrée analyzer [7,10].

Assuming we are interested in a loop-free program fragment, the above approach of keeping symbolic unions gives the same results as performing the analysis separately over every path in the fragment. A recent method for finding disjunctive loop invariants [16] is based on this idea: each path inside the loop body is considered separately. Two recent proposals use SMT-solving [20] as a decision procedure for the satisfiability of first-order arithmetic formulas in order to enumerate only paths that are needed for the progress of the analysis [13,25]. They can equivalently be seen as analyses over a multigraph of transitions between some distinguished control nodes. This multigraph has an exponential number of edges, but is never explicitly represented in memory; instead, this graph is *implicitly* or *succinctly* represented: its edges are enumerated as needed as solutions to SMT problems.

An additional claim of the methods that distinguish paths inside the loop body [16,25] is that they tend to generate better invariants than methods that do not, by behaving better with respect to the *widening operators* [9] used for enforcing convergence when searching for loop invariants by Kleene iterations. A related technique, *guided static analysis* [15], computes successive loop invariants for increasing subsets of the transitions taken into account, until all transitions are considered; again, the claim is that this approach avoids some gross overapproximation introduced by widenings.

All these methods improve the precision of the analysis by keeping the same abstract domain (say, convex polyhedra) but changing the operations applied and their ordering. An alternative is to change the abstract domain, for instance by moving from intervals to octagons or convex polyhedra [23], or the widening operator: for instance one may replace the classical widening on convex polyhedra [11,17] by an improved version [1], or use widening "up to", that is, use linear inequalities syntactically present in the program source code as possible widening steps [18].

There are many possible combinations of the above techniques, and it is not evident which ones perform more or less precisely or more or less efficiently on real-life examples. One needs to experiment. Unfortunately, the published literature on the subject lacks experimental comparative assessments. One purpose of this article is therefore to propose such experimental results.

This article makes the following contributions:

1. We recast the guided static analysis technique from [15] on the expanded multigraph from [25], considering entire paths instead of individual transitions, using SMT queries and binary decision diagrams. (See §3)
2. We improve the technique for obtaining disjunctive invariants from [16] by replacing the explicit exhaustive enumeration of paths by a sequence of SMT queries. (See §4)
3. We implemented these techniques, in addition to "classical" iterations and the original guided static analysis, inside a prototype static analyzer. This tool uses the LLVM bitcode format [21,22] as input, which can be produced by compilation from C, C++ and Fortran, enabling it to be run on many real-life programs. With respect to abstract domains on numerical variables, it uses the APRON library [19], which supports a variety of abstract domains from which we can choose with minimal changes to our analyzer.
   Our tool uses a *single static assignment* (SSA) intermediate representation, which allows us to perform cheap symbolic propagations and simplifications, described in Section 5.1.
4. We conducted extensive experiments with this tool, on real-life programs: (i) fixing the abstract domain, varying the iteration technique (§ 5.2) (ii) fixing the iteration technique, varying the abstract domain (§5.3).

2 Bases

2.1 Static analysis by abstract interpretation

Let X be the set of possible states of the program variables; for instance, if the program has 3 unbounded integer variables, then $X = \mathbb{Z}^3$. The set $\mathcal{P}(X)$ of subsets of X, partially ordered by inclusion, is the *concrete domain*. An *abstract domain* is a set $X^\sharp$ equipped with a partial order $\sqsubseteq$; for instance, it can be the domain of convex polyhedra in $\mathbb{Q}^3$ ordered by geometric inclusion. In this article, all abstract domains are supposed to contain machine-representable objects, and all $\sqsubseteq$ order relations are supposed to be decidable. The concrete and abstract domains are connected by a monotone *concretization* function $\gamma : \left(X^\sharp, \sqsubseteq\right) \to (\mathcal{P}(X), \subseteq)$: an element $x^\sharp \in X^\sharp$ represents a set $\gamma(x^\sharp)$.[5]

[5] Some presentations of abstract interpretation identify an element $x^\sharp$ with the set of states $\gamma(x^\sharp)$ that it represents. This leads to simpler notations, but may also confuse if, as in some of our constructions, there exist several $x^\sharp$ with identical $\gamma(x^\sharp)$, but distinct algorithmic behaviors.

We also assume a join operator $\sqcup : X^\sharp \times X^\sharp \to X^\sharp$, with infix notation; in practice, it is generally a least upper bound operation, but we only need it to satisfy $\gamma(x^\sharp) \cup \gamma(y^\sharp) \subseteq \gamma(x^\sharp \sqcup y^\sharp)$ for all $x^\sharp, y^\sharp$.

Classically, one considers the control-flow graph of the program, with edges labeled with concrete transition relations (e.g. $x' = x + 1$ for an instruction `x = x+1;`), and attaches an abstract element to each control point. A concrete transition relation $\tau \subseteq X \times X$ is replaced by an abstract *forward abstract transformer* $\tau^\sharp : X^\sharp \to X^\sharp$, such that

$$\forall x^\sharp \in X^\sharp, x, x' \in X,\ x \in \gamma(x^\sharp) \wedge (x, x') \in \tau \implies x' \in \gamma \circ \tau^\sharp(x^\sharp) \tag{1}$$

It is easy to see that if to any control point $p \in P$ we attach an abstract element $x^\sharp_p$ such that (i) for any p, $\gamma(x^\sharp_p)$ includes all initial states possible at control node p (ii) for any p, p', $\tau^\sharp_{p,p'}(x^\sharp_p) \sqsubseteq x^\sharp_{p'}$, noting $\tau_{p,p'}$ the transition from p to p', then $(\gamma(x^\sharp_p))_{p\in P}$ form an *inductive invariant*: by induction, when the control point is p, the program state always lies in $\gamma(x^\sharp_p)$.

Kleene iterations compute such an inductive invariant as the stationary limit, if it exists, of the following system: for each p, initialize $x^\sharp_p$ such that $\gamma(x^\sharp_p)$ is a superset of the initial states at point p; then iterate the following: if $\tau^\sharp_{p,p'}(x^\sharp_p) \not\sqsubseteq x^\sharp_{p'}$, replace $x^\sharp_{p'}$ by $x^\sharp_{p'} \sqcup \tau^\sharp_{p,p'}(x^\sharp_p)$. Such a stationary limit is bound to exist if $X^\sharp$ has no infinite ascending chain $a_1 \sqsubsetneq a_2 \sqsubsetneq \ldots$; this condition is however not met by domains such as intervals or convex polyhedra.

Widening-accelerated Kleene iterations proceed by replacing $x^\sharp_{p'} \sqcup \tau^\sharp_{p,p'}(x^\sharp_p)$ by $x^\sharp_{p'} \triangledown (x^\sharp_{p'} \sqcup \tau^\sharp_{p,p'}(x^\sharp_p))$ where $\triangledown$ is a *widening operator*: for all $x^\sharp, y^\sharp$, $\gamma(y^\sharp) \subseteq \gamma(x^\sharp \triangledown y^\sharp)$, and there exists no sequence $u^\sharp_1, u^\sharp_2, \ldots$ of the form $u^\sharp_{n+1} = u^\sharp_n \triangledown v^\sharp_n$ where $v^\sharp_n$ is another sequence. Such iterations necessarily lead to a stationary limit $(x^\sharp_p)_{p\in P}$, which defines an inductive invariant $(\gamma(x^\sharp_p))_{p\in P}$. Note that this invariant is not, in general, the least one expressible in the abstract domain, and may depend on the iteration ordering (the successive choices p, p').

Once an inductive invariant $\gamma((x^\sharp_p)_{p\in P})$ has been obtained, one can attempt *decreasing* or *narrowing* iterations to reduce it. In their simplest form, this just means running the following operation until a fixpoint or a maximal number of iterations are reached: for any p', replace $x^\sharp_{p'}$ by $x^\sharp_{p'} \sqcap \left(\bigsqcup_{p\in P} \tau^\sharp_{p,p'}(x^\sharp_p)\right)$. The result also defines an inductive invariant. These decreasing iterations are indispensable to recover properties from guards (tests) in the program in most iteration settings; unfortunately, certain loops, particularly those involving identity (no-operation) transitions, may foil them: the iterations immediately reach a fixpoint and do not decrease further (see example in §2.3). Sections 2.4 and 2.5 describe techniques that work around this problem.

Widening operations have a somewhat counterintuitive behavior: the result of an analysis (postcondition) is not necessarily monotonic with respect to the precondition. For instance, classical interval analysis with widening and decreasing iterations on the following loop yields a loop invariant $x \in [0, +\infty)$ if the precondition is $x \in [0, 0]$, but $x \in [0, 10]$ (strictly stronger) if the precondition is

$x \in [0, 10]$ (strictly weaker): **while** (x != 10) x=x+1;. This means that, when using widenings, being locally more precise (for instance, by using more expressive abstract domains, or widening operators that climb more slowly [1]) does not necessarily translate into higher final precision. It is almost always possible to concoct examples where a supposedly more precise abstraction (e.g. polyhedra) yields less precise results than a less precise one (e.g. intervals), but are these examples representative of real programs? This justifies our recourse to evaluation on realistic examples (§ 5.2).

2.2 SMT-solving

Boolean satisfiability (SAT) is the canonical NP-complete problem: given a propositional formula (e.g. $(a \vee \neg b) \wedge (\neg a \vee b \vee \neg c)$), decide whether it is satisfiable — and, if so, output a satisfying assignment. Despite an exponential worst-case complexity, the DPLL algorithm [20,6] solves many useful SAT problems in practice.

SAT was extended to *satisfiability modulo theory* (SMT): in addition to propositional literals, SMT formulas admit atoms from a theory. For instance, the theories of linear integer arithmetic (LIA) and linear real arithmetic (LRA) have atoms of the form $a_1x_1 + \cdots + a_nx_n \bowtie C$ where $a_1, \ldots, a_n, C$ are integer constants, $x_1, \ldots, x_n$ are variables (interpreted over $\mathbb{Z}$ for LIA and $\mathbb{R}$ or $\mathbb{Q}$ for LRA), and $\bowtie$ is a comparison operator $=, \neq, <, \leqslant, >, \geqslant$. Satisfiability for LIA and LRA is NP-complete, yet tools based on DPLL(T) approach [20,6] solve many useful SMT problems in practice. All these tools provide a *satisfying assignment* if the problem is satisfiable.

Most SMT solvers, including Z3, Yices,and all those supporting the full SMTLIB2 standard [5], offer an *incremental* interface: the client program specifies the formula as an initially empty conjunction, to which additional constraints are added, and calls a "check" function answering whether it is satisfiable; it may then backtrack some of the constraints and add other ones without restarting from scratch.

2.3 A simple, motivating example

Consider the following program, adapted from [25], where $\mathsf{input}(a, b)$ stands for a nondeterministic input in $[a, b]$:

```
1 extern int input();
2 void rate_limiter() {
3   int x_old = 0;
4   while (1) {
5     int x = input(-100000, 100000);
6     if (x > x_old+10) x = x_old+10;
7     if (x < x_old-10) x = x_old-10;
8     x_old = x;
9   }
10 }
```

This program implements a construct commonly found in control programs (in e.g. automotive or avionics): a rate limiter. For the sake of simplicity, we chose it to be fed on a nondeterministic input clamped between $[-100000, 100000]$, but in a real system it would be integrated in a reactive control loop and its input connected to a complex system with unknown output range.

The expected inductive invariant is $\mathsf{x\_old} \in [-100000, 100000]$, but classical abstract interpretation using intervals (or octagons or polyhedra) finds $\mathsf{x\_old} \in (-\infty, +\infty)$ [10]. Let us briefly see why.

Widening iterations converge to $\mathsf{x\_old} \in (-\infty, +\infty)$; let us now see why decreasing iterations fail to recover the desired invariant. The $\mathsf{x} > \mathsf{x\_old{+}10}$ test at line 6, if taken, yields $\mathsf{x\_old} \in (-\infty, 99990)$; followed by $\mathsf{x} = \mathsf{x\_old{+}10}$, we obtain $\mathsf{x} \in (-\infty, 100000)$, and the same after union with the no-operation "else" branch. Line 7 yields $\mathsf{x} \in (-\infty, +\infty)$.

We could use "widening up to" or "widening with thresholds", propagating the "magic values" ± 100000 associated to x into $\mathsf{x\_old}$, but these syntactic approaches cannot directly cope with programs for which $\mathsf{x} \in [-100000, +100000]$ is itself obtained by analysis. The guided static analysis of [15] does not perform better, and also obtains $\mathsf{x\_old} \in (-\infty, +\infty)$.

In contrast, let us distinguish all four possible execution paths through the tests at lines 6 and 7. The path through both "else" branches is infeasible; the program is thus equivalent to:

```
1 extern int input();
2 void rate_limiter() {
3   int x_old = 0;
4   while (1) {
5     int x = input(-100000, 100000);
6     if (x > x_old+10) x = x_old+10;
7     else if (x < x_old-10) x = x_old-10;
8     else x_old = x;
9   }
10 }
```

Classical interval analysis on this program yields $\mathsf{x\_old} \in [-100000, 100000]$. We have transformed the first program into the second, manually pruning out infeasible paths; yet in general the resulting program could be exponentially larger than the first (as in Fig. 1), even though not all feasible paths are needed to compute the invariant.

Following recent suggestions [13,25], we avoid this space explosion by keeping the second program implicit while simulating its analysis. This means we work on an implicitly represented transition multigraph (Fig. 1); it is succinctly representated by the transition graph of the first program. Our first contribution (§3) is to recast the "guided analysis" from [15] on such a succinct representation of the paths in lieu of the individual transitions. A similar explosion occurs in disjunctive invariant generation, following [16]; our second contribution (§4) applies our implicit representation to their method.

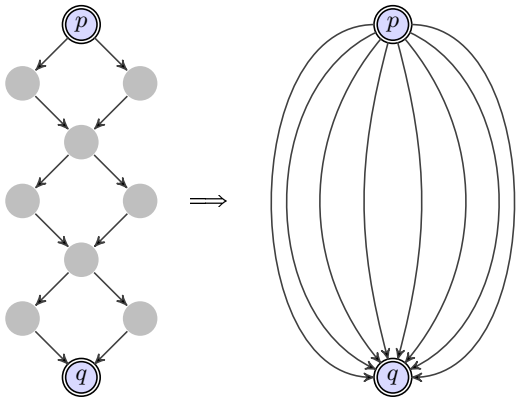

Fig. 1. Expansion of the transition graph into a multigraph.

2.4 Guided static analysis

Guided static analysis was proposed by [15] as an improvement over classical upward Kleene iterations with widening. Consider the following program, taken from [15]:

```
1 int x = 0;
2 int y = 0;
3 while (1) {
4   if (x <= 50) y++;
5   else y--;
6   if (y < 0) break;
7   x++;
8 }
```

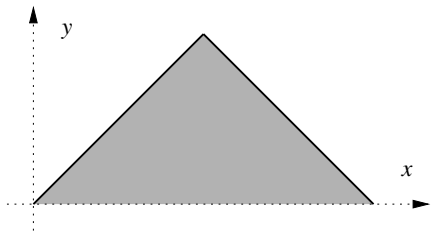

Fig. 2. The invariant for Prog. 2.4: the piecewise linear, solid line is the strongest invariant, the grayed polyhedron is its convex hull.

Classical iterations on the domain of convex polyhedra [11,1] or octagons [23] start with $x = 0 \wedge x = 0$, then continue with $x = y \wedge 0 \leqslant x \leqslant 1$. The widening

operator extrapolates from these two iterations and yields $x = y \wedge x \geqslant 0$. From there, the "else" branch at line 5 may be taken; with further widening, $0 \leqslant y \leqslant x$ is obtained as a loop invariant, and thus the postcondition computed at line 9 is $x \geqslant 0 \wedge y = 0$. Yet the strongest invariant is $(0 \leqslant x \leqslant 51 \wedge y = x) \vee (51 \leqslant x \leqslant 102 \wedge x + y = 102)$, and its convex hull, a convex polyhedron (Fig. 2), is

$$y \leqslant x \wedge y \leqslant 102 - x \wedge y \geqslant 0. \tag{2}$$

Intuitively, this disappointing result is obtained because widening extrapolates from the first iterations of the loop, but the loop has two different phases ($x \leqslant 50$ and $x > 50$) with different behaviors, thus the extrapolation from the first phase is not valid for the second.

Gopan and Reps' idea is to analyze the first phase of the loop with a widening and narrowing sequence, and thus obtain $0 \leqslant x \leqslant 50 \wedge y = x$, and then analyze the second phase, finally obtaining invariant 2; each phase is identified by the tests taken or not taken.

The analysis starts by identifying the tests taken and not taken during the first iteration of the loop, starting in the loop initialization. The branches not taken are pruned from the loop body, yielding:

```
while(1) {
  if(x <= 50) y++;
  else break; /* not taken in phase 1 */
  if(y < 0) break;
  x++;
}
```

Analyzing this loop using widening and narrowing on convex polyhedra or octagons yields the loop invariant $0 \leqslant x \leqslant 51 \wedge y = x$. Now, the transition at line 5 becomes feasible; and we analyze the full loop, starting iterations from $0 \leqslant x \leqslant 51 \wedge y = x$, and obtain invariant (2).

More generally, this analysis method considers an ascending sequence of subsets of the transitions in the loop body (left side of Fig. 1); for each subset, an inductive invariant is computed for the program restricted to it. The starting subset are the transitions reachable in one step from the loop initialization. If for a given subset S in the sequence, no transitions outside S are reachable from the inductive invariant attached to S, then iterations stop; otherwise, add these transitions to S and iterate more. Termination ensues from the finiteness of the control-flow graph.

2.5 Path-focusing

[25]'s *path-focusing* technique distinguishes the different paths in the program in order to avoid loss of precision due to merge operations. Since the number of paths may be exponential, the technique keeps them implicit and computes them when needed using SMT-solving. The (accelerated) Kleene iterations (§2.1) are computed over a reduced multigraph instead of the classical transition graph.

Let P be the set of control points in the transition graph, $P_W \subseteq P$ the set of widening points such that removing the points in P_W gives an acyclic graph. One can choose a set P_R such that $P_W \subseteq P_R \subseteq P$.

The set of paths is kept implicit by an SMT formula ρ expressing the semantics of the program, assuming that the transition semantics can be expressed within a decidable theory. For an easy construction of ρ, we also assume that the program is expressed in SSA form, meaning that each variable is only assigned once in the transition graph. This is not a restriction, since there exist standard algorithms that transform a program into an SSA format.

This formula contains Boolean *reachability predicates* b_i for each control points $p_i \notin P_R$, b_i^s and b_i^d for each $p_i \in P_R$, so that a path $p_{i_1} \to p_{i_2} \to \cdots \to p_{i_n}$ between two points $p_{i_1}, p_{i_n} \in P_R$ can easily be expressed as the conjunction $b_{i_1}^s \wedge \bigwedge_{2 \leqslant k < n} b_{i_k} \wedge b_{i_n}^d$. The Boolean b_i^s is *true* when the path starts at point p_i, whereas b_i^d is *true* when the path arrives at p_i. In other words, we split the points in P_R into a *source* point, with only outgoing transitions, and a *destination* point, with only incoming transitions, so that the resulting graph is acyclic and there is no paths going through control points in P_R.

In order to find focus paths, we solve an SMT formula which is satisfiable when there exist a path starting at a point $p_i \in P_R$ in a state included in the current invariant candidate X_i, and arriving at a point $p_j \in P_R$ in a state outside X_j. In this case, we construct this path using the model and update X_j. When $p_i = p_j$, meaning that the path is actually a self-loop, we can apply a widening/-narrowing sequence, or even compute the transitive closure of the loop (or an approximation thereof, or its application to X_i) using abstract acceleration [14].

3 Guided analysis over the paths

Guided static analysis, as proposed by [15], applies to the transition graph of the program. We now present a new technique applying this analysis on the implicit multigraph from [25], thus avoiding control flow merges with unfeasible paths. In this section, we use the same notations as 2.5, except that we call the abstract values X_i^s instead of X_i to leave room for another abstract value X_i^d later in the algorithm.

The combination of these two techniques aims at first discovering a precise inductive invariant for a subset of paths between two points in P_R, by the mean of ascending and narrowing iterations. When an inductive invariant has been found, we add new feasible paths to the subset and compute an inductive invariant for this new subset, starting with the results from the previous analysis. In other words, our technique considers an ascending sequence of subsets of the paths between two points in P_R. We iterate the operations until the whole program (i.e all the feasible paths) has been considered. The result will then be an inductive invariant of the entire program.

3.1 Algorithm

Algorithm 1 performs Guided static analysis on the implicitly represented multigraph.

The current working subset of paths, noted P and initially empty, is stored using a compact representation, such as binary decision diagrams. We also maintain two sets of control points:

- A' contains the points in P_R that may be the starting points of new feasible paths.
- A contains the points in P_R on which we apply the ascending iterations. Each time the abstract value of a control point p is updated, p is inserted in both A and A'.

Algorithm 1 Guided static analysis on implicit multigraph

```
1: A' ← {p|P_R/I_p ≠ ∅}
2: A ← ∅
3: P ← ∅  // Paths in the current subset
4: for all p_i ∈ P_R do
5:    X_i^s ← I_{p_i}
6: end for
7: while A' ≠ ∅ do
8:    P' ← ∅  // new paths
9:    for all p_i ∈ A' do
10:      ComputeNewPaths(p_i)  // Update A and P'
11:   end for
12:   P ← P ∪ P'
13:   A' ← ∅
14:   // ascending iterations on P
15:   while A ≠ ∅ do
16:      Select p_i ∈ A
17:      A ← A\{p_i}
18:      PathFocusing(p_i)  // Update A and A'
19:   end while
20:   Narrow
21: end while
22:
23: return {X_i^s, i ∈ P_R}
```

We distinguish three phases in the main loop of the analysis:

1. We start finding a new relevant subset $P \cup P'$ of the graph. Either the previous iteration or the initialization lead us to a state where there are no more paths in the previous subset P, starting at p_i, that make the abstract values of the successors grow (otherwise, the SMT solver would not have answered "*unsat*"). Narrowing iterations preserve this property. However,

there may exist such paths in the entire multigraph, that are not in P. This phase computes these paths and adds them to P'. This phase is described in 3.3 and corresponds to lines in 9 to 11 in Algorithm 1.

2. Given a new subset P, we search for paths starting at point $p_i \in P_R$, such that these paths are in P, i.e are included in the working subgraph. Each time we find a path, we update the abstract value of the destination point of the path. This is the phase explained in 3.2, and corresponds to lines 15 to 19 in Algorithm 1.
3. We perform narrowing iterations the usual way (line 20 in algorithm 1) and reiterate from step 1 unless there are no more points to explore, i.e. $A' = \varnothing$.

Step 2 corresponds to the application of path-focusing [25] to the elements of A until A is empty. When A becomes empty, it means the invariant computation of the subgraph is finished. As proposed by [15], we can do some narrowing iterations. These narrowing iterations allow to recover precision lost by widening, *before* computing and taking into account new feasible paths. Thus, our technique combines both the advantages of *Guided Static Analysis* and *Path-focusing*.

The order of steps is important: narrowing has to be performed before adding new paths, or some spurious new paths would be added to P, and starting with the addition of new paths avoids wasting time doing the ascending iterations on an empty graph.

3.2 Ascending iterations by Path-focusing

For computing an inductive invariant over a subgraph, we use the Path-focusing algorithm from [25] with special treatment for self loops (line 18 in algorithm 1).

In order to find which path to focus on, we construct an SMT formula $f(p_i)$, whose model when satisfiable is a path that starts in p_i, goes to a successor $p_j \in P_R$ of p_i, such that the image of X_i^s by the path transformation is not included in the current X_j^s. Intuitively, such a path makes the abstract value X_j^s grow, and thus is an interesting path to focus on. We loop until the formula becomes unsatisfiable, meaning that the analysis of p_i is finished.

If we note $Succ(i)$ the set of indices j such that $p_j \in P_R$ is a successor of p_i in the expanded multigraph, and X_i^s the abstract value associated to p_i :

$$f(p_i) = \rho \wedge b_i^s \wedge \bigwedge_{\substack{j \in P_R \\ j \neq i}} \neg b_j^s \wedge X_i^s \wedge \bigvee_{j \in Succ(i)} (b_j^d \wedge \neg X_j^s)$$

The difference with [25] is that we do not work on the entire transition graph but on a subset of it. Therefore we conjoin the formula $f(p_i)$ with the actual set of working paths, noted P, expressed as a Boolean formula, where the Boolean variables are the *reachability predicates* of the control points. We can easily construct this formula from the binary decision diagram using dynamic programming, and avoiding an exponentially sized formula. In other words, we force the SMT solver to give us a path included in P.

3.3 Adding new paths

Our technique computes the fixpoint iterations on a ascending sequence of subgraphs, until the complete graph is reached. When the analysis of a subgraph is finished, meaning that the abstract values for each control points has converged to an inductive invariant for this subgraph, the next subgraph to work on has to be computed.

This new subgraph is the union of the paths of the previous one with a set P' of new paths that become feasible regarding the current abstract values. The paths in P' are computed one after another, until no more path can make the invariant grow. This is line 10 in Algorithm 1, which corresponds to Algorithm 2. We also use SMT solving to discover these new paths, but we subtly change the SMT formula given to the SMT solver: we now simply check for the satisfiability of $f(p_i)$ instead of $f(p_i) \wedge P$. Since we already know that $f(p_i) \wedge P$ is unsatisfiable, none of the paths given by the SMT solver will be in P.

Also, to prevent the algorithm from adding too many paths at a time, and in particular to prevent the algorithm from trying to re-add the same path infinitely many times, we use another abstract value associated to the control point p_j, noted X_j^d, which is distinct from X_j^s, and initialized to X_j^s right before computing new paths. In the SMT formula, we associate to p_j the abstract value X_j^s when p_j is the starting point of the path, and X_j^d when it is its destination point. We thus check the satisfiability of the formula $f'(p_i)$, where:

$$f'(p_i) = \rho \wedge b_i^s \wedge \bigwedge_{\substack{j \in P_R \\ j \neq i}} \neg b_j^s \wedge X_i^s \wedge \bigvee_{j \in Succ(i)} (b_j^d \wedge \neg X_j^d)$$

X_j^d is updated when the point p_j is the target of a new path. This way, further SMT queries do not compute other paths with the same source and destination if it is not needed (because these new paths would not make X_j^d grow, hence would not be returned by the SMT solver).

Algorithm 2 ComputeNewPaths

```
1: while true do
2:    res ← SmtSolve[f'(p_i)]
3:    if res = unsat then
4:       break
5:    end if
6:    Compute the path e from the model
7:    Update X_j^d
8:    P' ← P' ∪ {e}
9:    A ← A ∪ {p_i}
10: end while
```

3.4 Termination

Termination of this algorithm is guaranteed, because:

- the subset of paths P strictly increases at each loop iteration, and bounded by the finite set of paths in the entire graph.
- the set P' always verifies $P \cap P' = \varnothing$ by construction, which guarantees that P' will eventually be empty after a finite number of loop iterations.

3.5 Example

We revise the rate limiter described in 2.3. In this example, *Path-focusing* works well because all the paths starting at the loop header are actually self loops. In such a case, the technique performs a widening/narrowing sequence or accelerates the loop, thus leading to a precise invariant. However, in some cases, there also exist paths that are not self loops, in which case *Path-focusing* applies widening. This widening may induce unrecoverable loss of precision.

Suppose the main loop of the rate limiter contains a nested loop:

```
1 extern int input();
2 extern int wait();
3 void rate_limiter() {
4   int x_old = 0;
5   while (1) {
6     int x = input(-100000, 100000);
7     if (x > x_old+10) x = x_old+10;
8     if (x < x_old-10) x = x_old-10;
9     x_old = x;
10    while (wait()) {}
11  }
12 }
```

We choose P_R as the set of loop headers of the function, plus the initial state. In this case, we have three elements in P_R. We also unroll the loop once, in order to distinguish the paths that do not go through the loop from the others:

```
if (wait()) {
   while (wait()) {}
}
```

The main loop in the expanded multigraph has then 8 distinct paths, half of them being self loops, and the other half going to the header of the nested loop.

Guided static analysis from [15] yields, at line 5, $\mathsf{x\_old} \in (-\infty, +\infty)$. Path-focusing [25] performs slightly better, and finds $\mathsf{x\_old} \in (-\infty, 10000]$. Now, let us see how our technique performs on this example.

Figure 3.5 shows the sequence of subset of paths during the analysis. The points in P_R are noted p_i, where i is the corresponding line in the code: for instance, p_5 corresponds to the header of the main loop.

1. The first subset of paths is depicted on Figure 3.5 Step 1. We apply Path-Focusing on this graph, and discover for p_5 the inductive invariant $\mathsf{x\_old} = 0$.

2. We then compute the set P' of paths that have to be added into the next subgraph. The image of $\mathsf{x\_old} = 0$ by the path that goes from p_5 to itself, and that goes through the *else* branch of each *if-then-else*, is $-10 \leqslant \mathsf{x\_old} \leqslant 10$. This path is then added to our subgraph. Moreover, there is no other path whose image is not in $-10 \leqslant \mathsf{x\_old} \leqslant 10$. Additionally, the path that goes from p_5 to p_{10}, that goes through the *else* branch of each *if-then-else* at lines 7,8 and 9, is also $-10 \leqslant \mathsf{x\_old} \leqslant 10$, and thus is also added into the new subgraph. Again, there is no new path that goes to p_{10} with an image outside $-10 \leqslant \mathsf{x\_old} \leqslant 10$.
3. We apply Path Focusing on this new subgraph (3.5 Step 2). When we have reached an inductive invariant, we narrow, and the result for both p_5 and p_{10} is the polyhedron $-10000 \leqslant \mathsf{x\_old} \leqslant 10000$.
4. Finally, we compute the new subgraph. The SMT-solver does not find any new path that makes the abstract values grow, and the algorithm terminates.

Our technique gives us the expected invariant $\mathsf{x\_old} \in [-10000, 10000]$. Here, only 3 paths out of the 8 have been computed during the analysis. In practice, depending on the order the SMT-solver returns the paths, other feasible paths could have been added during the analysis.

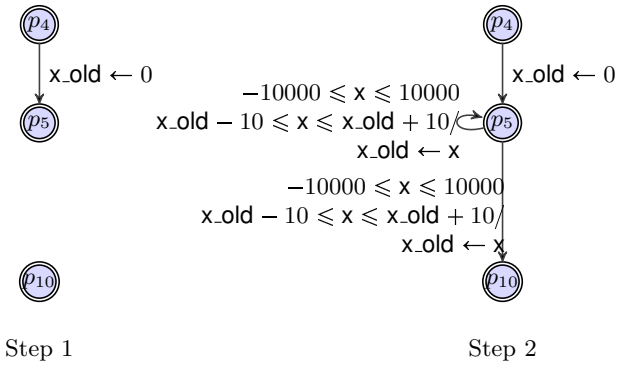

Fig. 3. Ascending sequence of subgraphs

In this example, we see that our technique actually combines best of *Guided Static Analysis* and *Path Focusing*. Still, we need to unroll the loop once in order to get precise results. Section 4 proposes a new technique that gives precise invariants without needing such graph transformations.

4 Disjunctive invariants

[16] proposed a technique for computing disjunctive invariants, by distinguishing all the paths inside a loop. In this section, we propose to improve this technique

by using SMT queries to find interesting paths, the objective being to avoid an explicit exhaustive enumeration of an exponential number of paths.

For each control point p_i, we compute a disjunctive invariant $\bigvee_{1 \leqslant j \leqslant m_i} X_{i,j}$. We denote by n_i the number of distinct paths starting at p_i. To perform the analysis, one chooses an integer $\delta_i \in [1, m_i]$, and a mapping function $\sigma_i : [1, m_i] \times [1, n_i] \mapsto [1, m_i]$. The k-th path starting fom p_i is denoted $\tau_{i,k}$. The image of the j-th disjunct $X_{i,j}$ by the path $\tau_{i,k}$ is then joined with $X_{i,\sigma(j,k)}$. Initially, the δ_i-th abstract value contains the initial states of p_i, and all other abstract values contain $\varnothing$.

For each control point $p_i \in P_R$, m_i, δ_i and σ_i can be defined heuristically. For instance, one could define σ_i so that $\sigma_i(j,k)$ only depends on the last transition of the path, or else construct it dynamically during the analysis.

Our method improves this technique in two ways :

- Instead of enumerating the whole set of paths, we keep them implicit and compute them only when needed.
- At each loop iteration of the original algorithm [16], an image by each path inside the loop is computed for each disjunct of the invariant candidate. Yet, many of these images may be redundant: for instance, if our invariant candidate is $(0 \leqslant x \leqslant 10 \wedge 0 \leqslant y \leqslant 1000) \vee (x < -10 \wedge y < -10)$, then there is no point enumerating paths whose image is included in this invariant candidate. In our approach, we compute such an image only if it makes the resulting abstract value grow.

Our improvement consists in a modification of the SMT formula we solve in 3. We introduce in this formula Boolean variables $\{d_j, 1 \leqslant j \leqslant m\}$, so that we can easily find in the model which abstract value of the disjunction of the source point has to be chosen to make the invariant of the destination grow. The resulting formula that is given to the SMT solver is defined by $g(p_i)$. When the formula is satisfiable, we know that the index j of the starting disjunct that has to be chosen is the one for which the associate Boolean value d_j is *true* in the model. Then, we can easily compute the value of $\sigma_i(j,k)$, thus know the index of the disjunct to join with.

$$g(p_i) = \rho \wedge b_i^s \wedge \bigwedge_{\substack{j \in P_R \\ j \neq i}} \neg b_j^s \wedge \bigvee_{1 \leqslant k \leqslant m_i} (d_k \wedge X_{i,k} \wedge \bigwedge_{l \neq k} \neg d_l) \wedge \bigvee_{j \in Succ(i)} (b_j^d \wedge \bigwedge_{1 \leqslant k \leqslant m_i} (\neg X_{j,k}))$$

In our algorithm, the initialization of the abstract values slightly differs from algorithm 1 line 5, since we now have to initialize each disjunct. Line 5 is then replaced by:

Furthermore, the Path-focused algorithm (line 18 from algorithm 1) is enhanced to deal with disjunctive invariants, and is detailed in algorithm 3.

The *Update* function can classically assign to $X_{i,\sigma_i(j,k)}$ the value $X_{i,\sigma_i(j,k)} \triangledown (X_{i,\sigma_i(j,k)} \sqcup \tau_{i,k}(X_{i,j}))$, or can integrate the special treatment for self loops proposed by [25], with widening/narrowing sequence or acceleration.

```
1: for all k ∈ {1, .., m_i}\{δ_i} do
2:     X_{i,k} ←⊥
3: end for
4: X_{i,δ_i} ← I_{p_i}
```

Algorithm 3 Disjunctive invariant computation with implicit paths

```
1: while true do
2:     res ← SmtSolve [g(p_i)]
3:     if res = unsat then
4:         break
5:     end if
6:     Compute the path τ_{i,k} from res
7:     Take j ∈ {l | d_l = true}
8:     Update(X_{i,σ_i(j,k)})
9: end while
```

We experimented with a heuristic of dynamic construction of the σ_i functions, adapted from [16]. For each control point $p_i \in P_R$, we start with one single disjunct ($m_i = 1$) and define $\delta_i = 1$. M denotes an upper bound on the number of disjuncts per control point.

The σ_i functions take as parameters the index of the starting abstract value, and the path we focus on. Since we dynamically construct these functions during the analysis, we store their already computed image into a compact representation, such as Algebraic Decision Diagrams. $\sigma_i(j,k)$ is then constructed on the fly only when needed, and computed only once. When the value of $\sigma_i(j,k)$ is required but undefined, we first compute the image of the abstract value $X_{i,j}$ by the path indexed by k, and try to find an existing disjunct of index j' so that the least upper bound of the two abstract values is exactly their union (this can be tested using SMT-solving). If such an index exists, then we set $\sigma_i(j,k) = j'$. Otherwise:

- if $m_i < M$, we increase m_i by 1 and define $\sigma_i(j,k) = m_i$
- if $m_i = M$, we define $\sigma_i(j,k) = M$

The main difference with the original algorithm [16] is that we construct $\sigma_i(j,k)$ using SMT queries instead of enumerating a possibly exponential number of paths to find a solution.

5 Experimental comparisons

We have implemented our proposed solutions inside a prototype of intraprocedural static analyzer, as well as the classical abstract interpretation algorithm, and the state-of-the-art techniques *Path Focusing* [25] and *Guided Static Analysis* [15].

Our tool operates over LLVM bitcode [22,21], which is a target for several compilers, most notably Clang (supporting C and C++) and llvm-gcc (supporting C, C++, Fortran and Ada). Abstract domains are provided by the APRON

library [19], and include convex polyhedra (from the builtin Polka "PK" library), octagons, intervals, and linear congruences. For SMT-solving, our analyzer uses Yices [12] or Microsoft Z3[27].

We conducted extensive experiments on real-life programs in order to compare the different techniques, mostly on open-source projects (Fig. 1) written in C, C++ and Fortran.

Name	kLOC	$\lvert P_R \rvert$
a2ps	55	2012
gawk	59	902
gnuchess	38	1222
gnugo	83	2801
grep	35	820
gzip	27	494
lapack/blas	954	16422
make	34	993
sed	23	292
tar	73	1712

Table 1. List of analyzed open-source projects, with their respective number of lines of code, and their number of control points in P_R

5.1 Analysis algorithm

For each program, we distinguish a set $P_R = P_W$ of suitable widening points by a simple algorithm: for each procedure, compute the strongly connected components of its control-flow graph using Tarjan's algorithm; the targets of the back-edges of the depth-first search are added to P_R. Note that the resulting set is not necessarily minimal, but is sufficient to disconnect all cycles — more sophisticated techniques are discussed in e.g. [8].

LLVM bitcode is in *static single assignment* (SSA) form: a given scalar variable is given a value at a single syntactic point in the program. In concrete terms, an assignment `x=2*x+1;` gets translated into a definition $x_2 = 2x_1 + 1$, with distinct variables x_1 and x_2 corresponding to the same original variable `x` at different points in the program. LLVM makes it easy to follow definition-use and use-definition chains: for a given variable (say, x_2) one can immediately obtain its definition (say, $2x_1 + 1$). One may see conversion to SSA form as a static precomputation of some of the symbolic propagations proposed by [24] to enhance the precision of analyses.

SSA introduces ϕ-functions at the head of a control code to define variables whose value depends on which incoming edge to this control node was last taken. For instance, for **if** (...) { `x = 2*x+1;` } **else** { `x= 0;` }, then x_2 is defined as $\phi(2x_1 + 1, 0)$. In this framework, each variable is uniquely defined as an arithmetic ($+$, $-$, $\times$, $/$) function of other variables that themselves may not

be representable as affine linear functions, because they are defined using ϕ-functions, bitwise arithmetic, loads from memory, or return values from function calls.

This motivates a key implement decision of our tool: only those variables $v_1, \ldots, v_n$ that are not defined by arithmetic operations are retained as coordinates in the abstract domain (e.g. as dimensions in polyhedra). A basic block of code therefore amounts to a *parallel assignment* operation $(v_1, \ldots, v_n) \mapsto (f_1(v_1, \ldots, v_n), , \ldots, f_n(v_1, \ldots, v_n))$; such operations are directly supported by APRON. This has three benefits: (i) it limits the number of dimensions in the abstract values, since polyhedra libraries typically perform worse with higher dimensions; (ii) the abstract operation for a single path in path-focusing methods also is a (large) parallel assignment; (iii) as suggested by [24], this approach is more precise than running abstract operations for each program line separately: for instance, for `y=x; z=x-y;` with precondition $x \in [0, 1]$, a line-by-line interval analysis obtains $y \in [0, 1]$ and $z \in [-1, 1]$ while our "en bloc" analysis symbolically simplifies $z = x - x = 0$ and thus $z \in [0, 0]$.

In the event that a node is reachable only by a single control-flow edge (which may occur because of dead code, or during the first phases of guided static analysis), the ϕ operation reduces to a copy of the values flowing from that edge. In this case, our tool just propagates symbolic values through the predecessor node, without introducing ϕ-variables.

We further reduce the number of variables by projecting out those that are not live at the program point. Again, SSA use-def and def-use chains make it easy to compute liveness.

Our tool currently only operates over scalar variables from the SSA representation and thus cannot directly cope with arrays or memory accessed through pointers. We therefore run it after the "memory to registers" (`mem2reg`) optimization phase in LLVM, which lifts most memory accesses to scalar variables. The remaining memory accesses are treated as undefined values, as are the return values of function calls. For better precision, we also apply function inlining to the program, and we unroll every loop once.

Our tool currently assumes that integer variables are unbounded mathematical integers ($\mathbb{Z}$) and floating-point variables are real (or rational) numbers. Techniques for sound analysis of bounded integers, including with wraparound, and of floating-point operations have been developed in e.g. the Astrée system [10,7], but porting these techniques to our iteration schemes using SMT-solving requires supplemental work.

Our implementation of path-focusing currently does not use true acceleration techniques, as proposed by [25]. Instead, it simply runs widening and narrowing iterations on a single path. The analysis is currently only forward, even though nothing in what we describe in this article is specific to forward analysis.

5.2 Precision of the various techniques

For each program and each pair (T_1, T_2) of analysis techniques, we list the proportion of control points in P_R where T_1 (resp. T_2) gives a strictly stronger

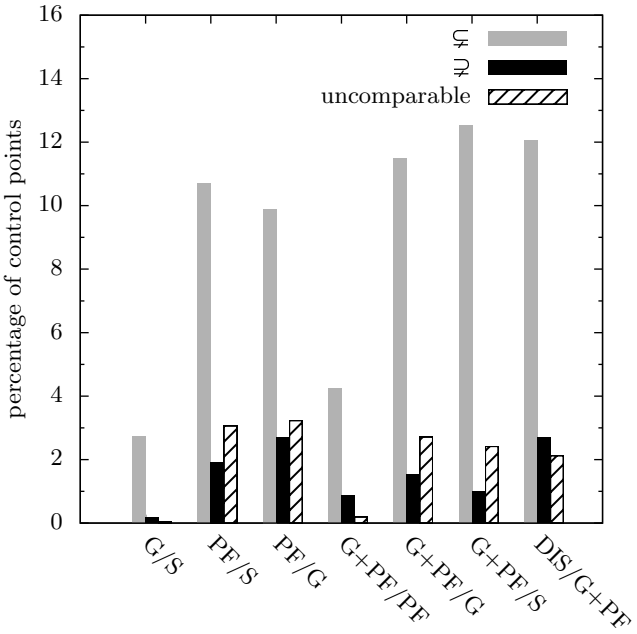

Fig. 4. Comparison of the abstract values obtained by classical abstract interpretation (S), *Guided Static Analysis* (G), *Path-focused* technique (PF), our combined technique (G+PF), and its version with disjunctive invariants (DIS). The $\subsetneq$ bars gives the percentage of invariants stronger (more precise; smaller with respect to inclusion) with the left-side technique, $\supsetneq$ the percentage of invariants stronger with the right-side technique, and "uncomparable" gives the percentage of invariants that are uncomparable, i.e neither greater nor smaller; the code points where both invariants are equal make up the remaining percentage. See Tab. 2 for details.

invariant, denoted by $\subsetneq$ (resp. $\supsetneq$), and the proportion of control points where the invariants given by T_1 and T_2 are uncomparable for the inclusion ordering (the remainder of the control points are thus those for which both techniques give the same invariant). We use convex polyhedra as the abstract domain.

Let us briefly comment the results given in more details in Table 2 and Figure 4. *Guided Static Analysis* from [15] improves the result of the classical Abstract Interpretation in 2.72% of the control points in P_R. *Path-focusing* from [25] gives statistically better results, and finds better invariants than *Guided Static Analysis* in 9.89% of the cases. However, it also loses precision in an important number (2.68%) of control points. Finally, our combined technique gives the most promising results, since it is statistically more precise than the other techniques.

The analysis using disjunctive invariants greatly improves the precision of the analysis (for 12.06% of the control points in P_R), at an acceptable time cost (see Table 3).

Benchmark	G/S			PF/S			PF/G			G+PF/PF			G+PF/G			DIS/G+PF		
	$\subsetneq$	$\supsetneq$	unc.	$\subsetneq$	$\supsetneq$	unc.	$\subsetneq$	$\supsetneq$	unc.	$\subsetneq$	$\supsetneq$	unc.	$\subsetneq$	$\supsetneq$	unc.	$\subsetneq$	$\supsetneq$	unc
a2ps-4.14	0.79	0	0	3.11	1.34	0.24	2.50	1.52	0.67	1.34	0.36	0.06	2.75	1.10	0.61	10.57	0.36	0.4
gawk-4.0.0	2.31	0	0	3.47	3.00	0.92	3.47	3.70	0.69	4.86	0	0	4.39	0.23	0.46	11.11	0	0.6
gnuchess-6.0.0	2.55	0.17	0	13.28	2.81	1.49	11.16	2.99	1.49	1.49	0.61	0	10.72	1.75	2.37	13.01	2.63	3.5
gnugo-3.8	1.84	0	0	13.34	2.27	2.64	12.23	2.52	2.70	2.21	0.49	0.92	13.16	2.27	2.21	10.14	1.23	1.8
grep-2.9	0.80	1.00	0	5.20	3.20	2.20	5.20	3.60	2.20	1.20	0.40	0	5.20	4.00	2.20	11.20	0.60	0.6
gzip-1.4	5.67	1.49	0.29	14.62	1.79	3.28	10.74	2.98	6.86	5.97	0.89	0.29	12.23	1.79	6.56	12.23	1.79	5.0
lapack-3.3.1	4.10	0.01	0	15.48	1.58	5.57	15.06	2.82	5.50	7.31	1.60	0.15	18.56	1.29	4.20	14.38	5.48	2.9
make-3.82	2.62	0	0	4.87	2.87	0.75	4.75	4.50	0.75	3.50	0.37	0.12	5.00	2.50	0.50	9.62	0.87	1.2
sed-4.2	1.73	1.73	0	5.55	1.04	1.73	3.81	1.04	1.73	2.43	0.34	0	3.81	1.04	1.73	6.59	0.69	2.0
tar-1.26	1.50	0.37	0.31	5.00	1.68	1.43	3.93	2.12	1.93	2.31	0.18	0	4.44	1.00	2.00	10.00	0.31	1.0

Table 2. Result of the comparison of the various techniques described in this paper: classic Abstract Interpretation (S), *Guided Static Analysis* (G), *Path-focusing* (PF), our combined technique (G+PF), and its version using disjunctive invariants (DIS). For instance, **G/S** compares the benefits of *Guided Static Analysis* over the classic Abstract interpretation algorithm. $\subsetneq$, $\supsetneq$ and "unc." are defined as in Fig. 4.

While experiencing with techniques that use SMT-solving, we encountered some limitations due to non-linear arithmetic in the analyzed programs. Indeed, the SMT-solver is not able to decide the satisfiability of some SMT-formulae expressing the semantics of non-linear programs. In this case, we skipped the functions for which the SMT-solver returned the "unknown" result. This limitation occurred very rarely in our experiments, except for the analysis of *Lapack/Blas*, where 798 over the 1602 functions have been skipped. *Lapack/Blas* implements matrix computations, which use floating-point multiplications.

In cases where the formula is expressed in too rich a logic for the SMT-solver to deal with, a number of workarounds are possible, including: (i) *Linearization*, as per [24], which overapproximates nonlinear semantics by linear semantics. (ii) Replacing the results of nonlinear operations by "unknown". Neither is currently implemented in our tool.

Table 3 gives the execution time of the different analysis techniques. It is interesting to see that *Guided Static Analysis* and *Path-focusing* are sometimes faster than the classical algorithm. This seems due to the fact that these algorithms start on a small subset of the graph, and then may need fewer iterations on the complete graph. The techniques we introduce in this article are slower, but speed is still on the same order of magnitude as the older ones.

5.3 Precision of Abstract Domains

For each program and each pair (D_1, D_2) of abstract domains, we compare by inclusion the invariants of the different control points in P_R (where P_R is defined as in 5.2). The main results are given in Table 4.

Benchmark	**S**	**G**	**PF**	**G+PF**	**DIS**
a2ps-4.14	64	17	66	175	512
gawk-4.0.0	59	8	18	43	129
gnuchess-6.0.0	112	21	122	374	866
gnugo-3.8	155	31	128	382	1403
grep-2.9	50	12	21	58	307
gzip-1.4	63	12	36	97	617
lapack-3.3.1	721	246	2690	6052	22222
make-3.82	22	7	32	83	255
sed-4.2	35	7	23	69	521
tar-1.26	270	39	110	304	1164

Table 3. Execution time for each technique, expressed in seconds

Statistically, the domain of convex polyhedra gives the better results, but commonly yields weaker invariants than the domains of octagons/intervals; this is a known weakness of its widening operator [26]. The Octagon domain appears to be much better than intervals; this is unsurprising since in most programs and libraries, bounds on loop indices are non constant: they depend on some parameters (array sizes etc.).

The Lapack/Blas benchmarks are unusual compared to the other programs. These libraries perform matrix computations, using nested loops over indices; such programs are the prime target for polyhedral loop optimization techniques and it is therefore unsurprising that polyhedra perform very well over them.

The analysis of linear equalities (PKEQ) performs very fast compared to other abstract domains, but yields very imprecise invariants: it only detects relations of the form $\sum_i a_i x_i = C$ where a_i and C are constants. The comparison between the domain of convex polyhedra and linear congruences shows the percentage of points where the use of congruences refines the invariant. This is the case for 1.04% of the control points in P_R.

Finally, we evaluated the benefits of the improved version of the widening operator for convex polyhedra from [1], compared to the classical widening. We found that the improved version from [1] yields more precise invariants for 6.58% of the control points in P_R.

6 Conclusion and future prospects

Roughly, an analysis by abstract interpretation is defined by the choice of an iteration strategy and an abstract domain. In this article, we demonstrated that changes in the iteration algorithm can significantly improve precision, sometimes while improving analysis times.

A common criticism of analysis techniques based on SMT-solving is that they do not scale up. Yet, our experiments show that, for numerical properties, they scale up to the size of typical functions and loops. It is however quite certain that, naively applied, they cannot scale to the kind of programs targeted by e.g.

Benchmark	PK/OCT			PK/BOX			OCT/BOX			PK/PKEQ			PK/GRID			POLY*/PC		
	$\subsetneq$	$\supsetneq$	unc.	$\subsetneq$	$\supsetneq$	unc.	$\subsetneq$	$\supsetneq$	unc.	$\subsetneq$	$\supsetneq$	unc.	$\subsetneq$	$\supsetneq$	unc.	$\subsetneq$	$\supsetneq$	u
a2ps-4.14	4.56	1.39	0.22	6.86	1.38	0.44	6.28	0	0	28.75	0	0	23.23	0	0.60	3.71	0.22	
gawk-4.0.0	6.49	0	0	9.09	0	0	6.49	0	0	27.27	0	0	26.15	0	1.11	3.52	0	
gnuchess-6.0.0	4.27	2.51	0.33	6.47	2.50	0.40	6.52	1.65	0.08	36.48	0	0	35.74	0	0.82	2.88	1.04	
gnugo-3.8	8.26	1.66	1.15	11.98	1.03	1.54	11.53	0.21	0.21	34.52	0	0	36.46	0	0.44	8.93	1.23	
grep-2.9	1.60	1.77	0.88	4.98	1.06	1.60	5.69	0	0	27.04	0	0	26.86	0	0.17	2.86	0	
gzip-1.4	7.14	1.72	0.24	10.83	0.73	1.23	9.60	0	0.24	32.69	0	0	32.45	0	0.24	6.15	1.47	
lapack-3.3.1	24.96	1.61	5.13	54.06	0.57	5.99	58.42	0.48	1.02	64.14	0.06	0.06	62.50	0	1.63	9.55	0.24	
make-3.82	2.44	2.11	0.22	7.03	0.97	1.19	7.11	0.11	0	31.11	0	0	32.78	0	0.82	4.00	0.33	
sed-4.2	0.34	0.34	1.73	1.73	0.34	1.73	3.47	0	0	11.11	0	1.73	8.09	0	2.85	2.08	0	
tar-1.26	1.92	0.72	0.06	4.85	0.59	0.35	4.85	0.11	0	25.11	0	0.05	24.57	0	0.70	1.71	0.23	

Table 4. Results of the comparison of the various abstract domains, when using the same technique (G+PF). We used as abstract domains Convex Polyhedra (PK and POLY), Octagons (OCT), intervals (BOX), linear equalities (PKEQ) and linear congruences (GRID). The last column compares the domain of Convex Polyhedra with the improved widening operator from [1] (POLY*), and Convex Polyhedra using the classical widening operator (POLY). POLY and POLY* use the Parma Polyhedral Library [2]. $\subsetneq$, $\supsetneq$ and "unc." are defined as in Fig. 4.

the Astrée tool, that is, a dozens or hundreds of thousands of lines of code in a single loop operating over similar numbers of remanent variables. Actually, for such applications, only (quasi-)linear algorithms scale up, and "cheap" abstract domains such as octagons ($O(n^3)$ where n is the number of variables) are not applied to the full variable set, but to restricted subsets thereof. It thus seems reasonable that techniques such as considering "packs" of related variables, slicing, etc. may similarly help SMT-based techniques to scale to global analyses.

We compared the precision of different techniques and abstract domains by comparing the invariants for the inclusion ordering. A better metric is perhaps to take a client analysis — such as the detection of overflows and array bound violations — and compare the rates of alarms.

We focused on numerical properties, because they are supported by easily available abstract libraries. Yet, in most programs, properties of data structures are important for proving interesting properties. Further investigations are needed not only on good abstractions for pointers (many are already known) but also on their conversion to SMT problems.

References

1. Bagnara, R., Hill, P.M., Ricci, E., Zaffanella, E.: Precise widening operators for convex polyhedra. Science of Computer Programming 58(1–2), 28–56 (Oct 2005)
2. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library, version 0.9, `http://www.cs.unipr.it/ppl`

3. Bagnara, R., Hill, P.M., Zaffanella, E.: Widening operators for powerset domains. International Journal on Software Tools for Technology Transfer (STTT) 8(4-5), 449–466 (Aug 2006)
4. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming 72(1–2), 3–21 (2008)
5. Barrett, C., Stump, A., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB). `www.SMT-LIB.org` (2010)
6. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam (2009)
7. 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)
8. Bourdoncle, F.: Sémantiques des Langages Impératifs d'Ordre Supérieur et Interprétation Abstraite. Ph.D. thesis, École Polytechnique (1992), `http://web.me.com/fbourdoncle/page4/page15/files/dissertation.pdf`
9. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. of Logic and Computation pp. 511–547 (Aug 1992)
10. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: Sagiv, S.M. (ed.) Programming Languages and Systems (ESOP). pp. 21–30. No. 3444 in Lecture Notes in Computer Science, Springer Verlag (2005)
11. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Principles of Programming Languages (POPL). pp. 84–96. ACM (1978)
12. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV. Lecture Notes in Computer Science, vol. 4144, pp. 81–94. Springer (2006)
13. Gawlitza, T., Monniaux, D.: Improving strategies via SMT solving. In: Barthe, G. (ed.) ESOP. pp. 236–255. No. 6602 in Lecture Notes in Computer Science, Springer Verlag (2011)
14. Gonnord, L., Halbwachs, N.: Combining widening and acceleration in linear relation analysis. In: Yi, K. (ed.) Static analysis (SAS). Lecture Notes in Computer Science, vol. 4134, pp. 144–160. Springer Verlag (2006)
15. Gopan, D., Reps, T.W.: Guided static analysis. In: Nielson, H.R., Filé, G. (eds.) SAS. Lecture Notes in Computer Science, vol. 4634, pp. 349–365. Springer (2007)
16. Gulwani, S., Zuleger, F.: The reachability-bound problem. In: Zorn, B.G., Aiken, A. (eds.) PLDI. pp. 292–304. ACM (2010)
17. Halbwachs, N.: Détermination automatique de relations linéaires vérifiées par les variables d'un programme. Ph.D. thesis, Grenoble University (1979)
18. Halbwachs, N., Proy, Y.E., Roumanoff, P.: Verification of real-time systems using linear relation analysis. Formal Methods in System Design 11(2), 157–185 (August 1997)
19. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV. Lecture Notes in Computer Science, vol. 5643, pp. 661–667. Springer Verlag (2009)
20. Kroening, D., Strichman, O.: Decision procedures. Springer Verlag (2008)
21. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the international symposium on Code

generation and optimization: feedback-directed and runtime optimization. pp. 75–86. CGO '04, IEEE Computer Society, Washington, DC, USA (Aug 2004)
22. LLVM team: LLVM Language Reference Manual (2011), http://llvm.org/docs/LangRef.html
23. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
24. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI. Lecture Notes in Computer Science, vol. 3855, pp. 348–363. Springer (2006)
25. Monniaux, D., Gonnord, L.: Using bounded model checking to focus fixpoint iterations. In: Yahav, E. (ed.) Static analysis (SAS). Lecture Notes in Computer Science, vol. 6887, pp. 369–385. Springer Verlag (2011)
26. Monniaux, D., Le Guen, J.: Stratified static analysis based on variable dependencies (2011), http://arxiv.org/abs/1109.2405
27. de Moura, L.M., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
28. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. Transactions on Programming Languages and Systems (TOPLAS) 29(5), 26 (2007)