

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.

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 [8]. 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) [10,16,2,4].

In such an analysis, information may flow forward 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 co-reachable 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.

^{*}  This work was partially funded by ANR project “ASOPT”.

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* [29] implemented in the Astrée analyzer [7,9].

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 [15] is based on this idea: each path inside the loop body is considered separately. Two recent proposals use SMT-solving [22] 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 [12,27]. 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 in favor of the methods that distinguish paths inside the loop body [15,27] is that they tend to generate better invariants than methods that do not, by behaving better with respect to the *widening operators* [8] used for enforcing convergence when searching for loop invariants by Kleene iterations. A related technique, *guided static analysis* [14], 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 over-approximation 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 (e.g. octagons, convex polyhedra [25]), or the widening operator [1,17].

This article makes the following contributions:

1. We recast the guided static analysis technique from [14] on the expanded multigraph from [27], 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 [15] 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 [23,24] as input, which can be produced by compilation from C, C++ and Fortran, enabling it to be run on many real-life programs. It uses the APRON library [21], which supports a variety of abstract domains for numerical variables, from which we can choose with minimal changes to our analyzer.

4. We conducted extensive experiments with this tool, on real-life programs.

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 (the associated strict order being \sqsubset); for instance, it can be the domain of convex polyhedra in \mathbb{Q}^3 ordered by geometric inclusion. The concrete and abstract domains are connected by a monotone *concretization* function $\gamma : (X^\sharp, \sqsubseteq) \rightarrow (\mathcal{P}(X), \sqsubseteq)$: an element $x^\sharp \in X^\sharp$ represents a set $\gamma(x^\sharp)$.

We also assume a join operator $\sqcup : X^\sharp \times X^\sharp \rightarrow X^\sharp$, 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 $\mathbf{x} = \mathbf{x} + \mathbf{1}$); 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* $\tau^\sharp : X^\sharp \rightarrow X^\sharp$, such that $\forall x^\sharp \in X^\sharp, x, x' \in X, x \in \gamma(x^\sharp) \wedge (x, x') \in \tau \implies x' \in \gamma \circ \tau^\sharp(x^\sharp)$. It is easy to see that if to any control point $p \in P$ we attach an abstract element x_p^\sharp such that (i) for any p , $\gamma(x_p^\sharp)$ includes all initial states possible at control node p (ii) for any $p, p', \tau_{p,p'}^\sharp(x_p^\sharp) \sqsubseteq x_{p'}^\sharp$, noting $\tau_{p,p'}$ the transition from p to p' , then $(\gamma(x_p^\sharp))_{p \in P}$ form an *inductive invariant*: by induction, when the control point is p , the program state always lies in $\gamma(x_p^\sharp)$.

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

Widening-accelerated Kleene iterations proceed by replacing $x_{p'}^\sharp \sqcup \tau_{p,p'}^\sharp(x_p^\sharp)$ by $x_{p'}^\sharp \nabla (x_{p'}^\sharp \sqcup \tau_{p,p'}^\sharp(x_p^\sharp))$ where ∇ is a *widening operator*: for all $x^\sharp, y^\sharp, \gamma(y^\sharp) \subseteq \gamma(x^\sharp \nabla y^\sharp)$, and any sequence $u_1^\sharp, u_2^\sharp, \dots$ of the form $u_{n+1}^\sharp = u_n^\sharp \nabla v_n^\sharp$, where v_n^\sharp is another sequence, become stationary. The stationary limit $(x_p^\sharp)_{p \in P}$, defines an inductive invariant $(\gamma(x_p^\sharp))_{p \in P}$. 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 $\gamma((x_p^\#)_{p \in P})$ 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 $x_{p'}^\#$ by $x_{p'}^\# \cap \left(\bigsqcup_{p \in P} \tau_{p,p'}^\#(x_p^\#) \right)$. 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.

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 [22,6] solves many useful SAT problems in practice.

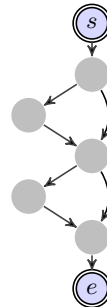
SAT was extended to *satisfiability modulo theory* (SMT): in addition to propositional literals, SMT formulas admit atoms from a theory. For instance, the theories of linear integer arithmetic (LIA) and linear real arithmetic (LRA) have atoms of the form $a_1 x_1 + \dots + a_n x_n \bowtie C$ where a_1, \dots, a_n, C are integer constants, x_1, \dots, x_n are variables (interpreted over \mathbb{Z} for LIA and \mathbb{R} or \mathbb{Q} for LRA), and \bowtie is a comparison operator $=, \neq, <, \leq, >, \geq$. Satisfiability for LIA and LRA is NP-complete, yet tools based on DPLL(T) approach [22,6] solve many useful SMT problems in practice. All these tools provide a *satisfying assignment* if the problem is satisfiable.

2.3 A Simple, Motivating Example

Consider the following program, adapted from [27], where `input(a, b)` stands for a nondeterministic input in $[a, b]$ (the control-flow graph on the right depicts the loop body, s is the start node and e the end node):

```

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



This program implements a construct commonly found in control programs (in e.g. automotive or avionics): a rate or slope limiter.

The expected inductive invariant is $\mathbf{x_old} \in [-100000, 100000]$, but classical abstract interpretation using intervals (or octagons or polyhedra) finds $\mathbf{x_old} \in (-\infty, +\infty)$ [9]. Let us briefly see why.

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

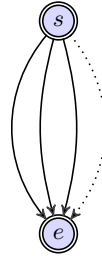
We could use “widening up to” or “widening with thresholds”, propagating the “magic values” ± 100000 associated to \mathbf{x} into $\mathbf{x_old}$, but these syntactic approaches cannot directly cope with programs for which $\mathbf{x} \in [-100000, +100000]$ is itself obtained by analysis. The guided static analysis of [14] does not perform better, and also obtains $\mathbf{x_old} \in (-\infty, +\infty)$.

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 a program with 3 paths:

```

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

```



Classical interval analysis on this program yields $\mathbf{x_old} \in [-100000, 100000]$. We have transformed the program, manually pruning out infeasible paths; yet in general the resulting program could be exponentially larger than the first, even though not all feasible paths are needed to compute the invariant.

Following recent suggestions [12,27], 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; it is succinctly represented by the transition graph of the first program. Our first contribution (§3) is to recast the “guided analysis” from [14] on such a succinct representation of the paths in lieu of the individual transitions. A similar explosion occurs in disjunctive invariant generation, following [15]; our second contribution (§4) applies our implicit representation to their method.

2.4 Guided Static Analysis

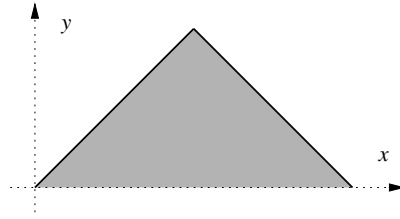
Guided static analysis was proposed by [14] as an improvement over classical upward Kleene iterations with widening. Consider the program in Fig. 1, taken from [14].

Classical iterations on the domain of convex polyhedra [10,1] or octagons [25] start with $x = 0 \wedge x = 0$, then continue with $x = y \wedge 0 \leq x \leq 1$. The widening operator extrapolates from these two iterations and yields $x = y \wedge x \geq 0$. From

```

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

```



$$y \leq x \wedge y \leq 102 - x \wedge y \geq 0.$$

Fig. 1. Example program and its invariant: the piecewise linear, solid line is the strongest invariant, the grayed polyhedron is its convex hull

there, the “else” branch at line 4 may be taken; with further widening, $0 \leq y \leq x$ is obtained as a loop invariant, and thus the computed loop postcondition is $x \geq 0 \wedge y = 0$. Yet the strongest invariant is $(0 \leq x \leq 51 \wedge y = x) \vee (51 \leq x \leq 102 \wedge x + y = 102)$, and its convex hull, a convex polyhedron (Fig. 1).

Intuitively, this disappointing result is obtained because widening extrapolates from the first iterations of the loop, but the loop has two different phases ($x \leq 50$ 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 $0 \leq x \leq 50 \wedge y = x$, and then analyze the second phase, finally obtaining invariant (2.4); 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 $0 \leq x \leq 51 \wedge y = x$. Now, the transition at line 4 becomes feasible; and we analyze the full loop, starting iterations from $0 \leq x \leq 51 \wedge y = x$, and obtain invariant (2.4) in Fig 1.

More generally, this analysis method considers an ascending sequence of subsets of the transitions in the loop body ; for each subset, an inductive invariant is computed for the program restricted to it. The starting subset consists in 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

Monniaux & Gonnord’s *path-focusing* [27] 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 exists 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 $p_{i_1} \rightarrow p_{i_2} \rightarrow \dots \rightarrow p_{i_n}$ between two points $p_{i_1}, p_{i_n} \in P_R$ can easily be expressed as the conjunction $b_{i_1}^s \wedge \bigwedge_{2 \leq k < n} b_{i_k} \wedge b_{i_n}^d$. 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 are 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 exists 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 [13].

We assume that we can encode the concrete semantics of the program into the SMT formula, or at least an abstraction thereof at least as precise as the one applied by the abstract interpreter (in simple terms: we want to avoid the case where the SMT solver exhibits a possible path, but the static analyzer realizes that this path is infeasible; this would lead to nontermination, because the SMT solver would exhibit the same path on the next iteration). A workaround would be to apply *satisfiability modulo path programs* [18]: from each path ruled infeasible by abstract interpretation, extract a blocking clause for the SAT solver underlying the SMT-solver.

3 Guided Analysis over the Paths

Guided static analysis, as proposed by [14], applies to the transition graph of the program. We now present a new technique applying this analysis on the implicit

multigraph from [27], thus avoiding control flow merges with unfeasible paths. In this section, we use the same notations as §2.5.

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.

The ascending iteration applies path-focusing [27] to a subset of the multigraph. As [14], we do some narrowing, to recover precision lost by widening, *before* computing and taking into account new feasible paths. Thus, our technique combines the advantages of *Guided Static Analysis* and *Path-focusing*.

Algorithm 1 performs Guided static analysis on the implicitly represented multigraph. I_p denotes a set of initial states at program point p (thus \emptyset for most p). 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' : points in P_R that may be the starting points of new feasible paths.
- A : points in P_R on which we apply the ascending iterations. When the abstract value of a control point p is updated, p is added to both A and A' .

Algorithm 1. Guided static analysis on implicit multigraph

```

1:  $A' \leftarrow \{p \mid P_R/I_p \neq \emptyset\}$ 
2:  $A \leftarrow \emptyset$ 
3:  $P \leftarrow \emptyset$  // Paths in the current subset
4: for all  $p_i \in P_R$  do
5:    $X_i \leftarrow I_{p_i}$ 
6: end for
7: while  $A' \neq \emptyset$  do
8:   while  $A' \neq \emptyset$  do
9:     Select  $p_i \in A'$ 
10:     $A' \leftarrow A' \setminus \{p_i\}$ 
11:    ComputeNewPaths( $p_i$ ) // Update  $A$ ,  $A'$  and  $P$ 
12:   end while
13:   // ascending iterations on  $P$ 
14:   while  $A \neq \emptyset$  do
15:     Select  $p_i \in A$ 
16:      $A \leftarrow A \setminus \{p_i\}$ 
17:     PathFocusing( $p_i$ ) // Update  $A$  and  $A'$ 
18:   end while
19:   Narrow
20: end while
21: return  $\{X_i, i \in P_R\}$ 

```

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

1. We start finding a new relevant subset P of the graph. Either the previous iteration or the initialization led 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 the subset. This phase is described in 3.2 and corresponds to lines in 8 to 12 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 3.1, and corresponds to lines 14 to 18 in Algorithm 1.
3. We perform narrowing iterations the usual way (line 19 in algorithm 1) and reiterate from step 1 unless there are no more points to explore, i.e. $A' = \emptyset$.

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

3.1 Ascending Iterations by Path-focusing

For computing an inductive invariant over a subgraph, we use the Path-focusing algorithm from [27] with special treatment for self loops (line 17 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 by the path transformation is not included in the current X_j . Intuitively, such a path makes the abstract value X_j 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 the abstract value associated to p_i :

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

The difference with [27] 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 . Each time the invariant candidate of a point p_j has been updated, p_j is inserted into A' since it may be the start of a new feasible paths.

3.2 Adding New Paths

Our technique computes the fixpoint iterations on an 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 point has converged to an inductive invariant for this subgraph, the next subgraph to work on has to be computed.

This new subgraph contains all the paths from the previous one, and also new paths that become feasible regarding the current abstract values. The new paths in P are computed one after another, until no more path can make the invariant grow. This is line 11 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 try to find a path that is not yet in P , but is feasible and makes the invariant candidate of its destination grow. We thus check the satisfiability of the formula $f'(p_i)$, where:

$$f'(p_i) = f(p_i) \wedge \neg P$$

X_j is updated using an abstract union 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 grow, hence would not be returned by the SMT solver).

Algorithm 2. ComputeNewPaths

```

1: while true do
2:    $res \leftarrow SmtSolve[f'(p_i)]$ 
3:   if  $res = unsat$  then
4:     break
5:   end if
6:   Compute the path  $e$  from the model
7:    $X_j \leftarrow X_j \sqcup \tau_e(X_i)$ 
8:    $P \leftarrow P \cup \{e\}$ 
9:    $A \leftarrow A \cup \{p_i\}$ 
10:   $A' \leftarrow A' \cup \{p_i\}$ 
11: end while

```

When a new path has been found, it is immediately added into P . We then have to add p_i and p_j into A (since we do not apply widening in this section) and p_j into A' , since p_j may be the starting point of a new feasible path.

3.3 Termination

Termination of this algorithm is guaranteed, because: 1. the subset of paths P strictly increases at each loop iteration, and is bounded by the finite set of paths in the entire graph. 2. when computing new paths, we conjunct our formula with $\neg P$, meaning that we obtain each possible path only once. The number of path is finite, so this computation always terminates. 3. the Path-focusing iterations terminate because of the properties of widening.

3.4 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 exists 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 like:

```

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

```

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 .

The main loop in the expanded multigraph has then 4 distinct paths going to the header of the nested loop.

Guided static analysis from [14] yields, at line 3, $\mathbf{x_old} \in (-\infty, +\infty)$. Path-focusing [27] also finds $\mathbf{x_old} \in (-\infty, +\infty)$. Now, let us see how our technique performs on this example.

Figure 2 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_3 corresponds to the header of the main loop.

1. The starting subgraph is depicted on Figure 2 Step 1. At the beginning, this graph has no transitions.
2. We compute the new feasible paths that have to be added into the subgraph. We first find the path from p_1 to p_3 and obtain at p_3 $\mathbf{x_old} = 0$.

The image of $\mathbf{x_old} = 0$ by the path that goes from p_3 to p_8 , and that goes through the *else* branch of each *if-then-else*, is $-10 \leq \mathbf{x_old} \leq 10$. This path is then added to our subgraph.

Moreover, there is no other path starting at p_3 whose image is not in $-10 \leq \mathbf{x_old} \leq 10$.

Finally, since the abstract value associated to p_8 is $-10 \leq \mathbf{x_old} \leq 10$, the path from p_8 to p_3 is feasible and is added into P . The final subgraph is depicted on Figure 2 Step 2.

3. We then compute the ascending iterations by path-focusing. At the end of these iterations, we obtain $-\infty \leq \mathbf{x_old} \leq +\infty$ for both p_3 and p_8 .
4. We now can apply narrowing iterations, and recover the precision lost by widening: we obtain $-10000 \leq \mathbf{x_old} \leq 10000$ at points p_3 and p_8 .
5. Finally, we compute the next 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 $\mathbf{x_old} \in [-10000, 10000]$. Here, only 3 paths out of the 6 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.

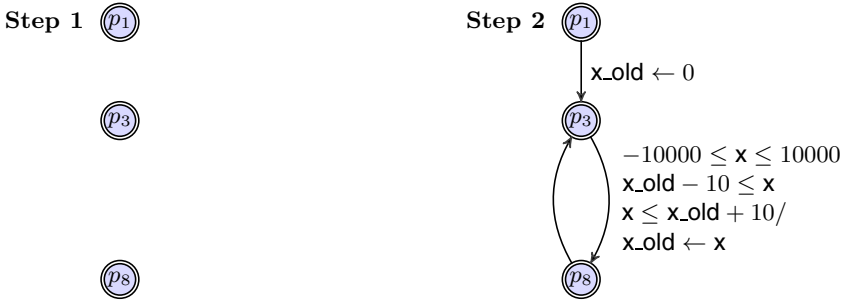


Fig. 2. Ascending sequence of subgraphs

In this example, we see that our technique actually combines best of *Guided Static Analysis* and *Path Focusing*.

4 Disjunctive Invariants

While many (most?) useful program invariants on numerical variables can be expressed as conjunctions of inequalities and congruences, it is sometimes necessary to introduce disjunctions. For instance, the loop **for** (**int** $i=0$; $i < n$; $i++$) $\{ \dots \}$ has head invariant $0 \leq i \leq n \vee (i = 0 \wedge n < 0)$. For this very simple example, a simple syntactic transformation of the control structure (into $i=0$; **if** ($i < n$) **do** $\{ \dots \}$ **while** ($i < n$)) is sufficient, but in more complex cases more advanced analyses are necessary [5,20,30,26]; in intuitive terms, they discover *phases* or *modes* in loops.

Gulwani & Zuleger [15] 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 $\bigvee_{1 \leq j \leq m_i} X_{i,j}$. 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 from 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 $X_{i,\sigma_i(j,k)}$. Initially, the δ_i -th abstract value contains the initial states of p_i , and all other abstract values contain \emptyset .

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 [15], 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 $(0 \leq x \leq 10 \wedge 0 \leq y \leq 1000) \vee (x < -10 \wedge y < -10)$, 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 $\{d_j, 1 \leq j \leq m\}$, 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.

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

In our algorithm, the initialization of the abstract values slightly differs from algorithm 1 line 5, since we now have to initialize each disjunct. Instead of Line 5, we initialize $X_{i,k}$ with \perp for all $k \in \{1, \dots, m_i\} \setminus \{\delta_i\}$, and X_{i,δ_i} with $\leftarrow I_{p_i}$.

Furthermore, the Path-focused algorithm (line 17 from algorithm 1) is enhanced to deal with disjunctive invariants, and is detailed in algorithm 3.

The *Update* function can classically assign $X_{i,\sigma_i(j,k)} \nabla (X_{i,\sigma_i(j,k)} \sqcup \tau_{i,k}(X_{i,j}))$ to $X_{i,\sigma_i(j,k)}$, or can integrate the special treatment for self loops proposed by [27], with widening/narrowing sequence or acceleration.

We experimented with a heuristic of dynamic construction of the σ_i functions, adapted from [15]. 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 (using SMT-solving). If such an index exists, then we set $\sigma_i(j, k) = j'$. Otherwise:

Algorithm 3. Disjunctive invariant computation with implicit paths

```

1: while true do
2:    $res \leftarrow SmtSolve[g(p_i)]$ 
3:   if  $res = unsat$  then
4:     break
5:   end if
6:   Compute the path  $\tau_{i,k}$  from  $res$ 
7:   Take  $j \in \{l \mid d_l = true\}$ 
8:   Update( $X_{i,\sigma_i(j,k)}$ )
9: end while

```

- 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 [15] 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 Implementation and Experimental Comparisons

We have implemented our proposed solutions inside a prototype of intraprocedural static analyzer called PAGAI, as well as the classical abstract interpretation algorithm, and the state-of-the-art techniques *Path Focusing* [27] and *Guided Static Analysis* [14]. It is available online at <https://forge.imag.fr/projects/pagai/>. The implementation is documented in [19].

PAGAI operates over LLVM bitcode [24,23], 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 [21], and include convex polyhedra (from the builtin Polka “PK” library), octagons, intervals, and linear congruences. For SMT-solving, our analyzer uses Yices [11] or Microsoft Z3 [28].

PAGAI currently neither models the memory heap nor performs interprocedural analysis. Instead, LLVM optimization phases are applied prior to analysis, in order to inline non-recursive function calls and lift certain memory accesses to operations on explicit numerical variables (e.g. $y=t[0]*t[0]$; preceded by $t[0]=x$; without any aliased write in between is replaced by $y=x*x$). The remaining memory reads are considered as indeterminates, and memory writes are ignored; this is a sound abstraction.

We conducted extensive experiments on real-life programs in order to compare the different techniques, mostly on open-source projects (Fig. 3) written in C, C++ and Fortran. These results confirm that our combined technique improve the analysis in comparison with the two techniques taken individually, at a reasonable cost. The extension with disjunctive invariants increases precision in many cases, but with higher cost in terms of execution time.

Table 1. Execution times for various techniques

Name	Size		Execution time (seconds)				
	kLOC	$ P_R $	S	G	PF	G+PF	DIS
a2ps-4.14	55	2012	23	74	34	115	162
gawk-4.0.0	59	902	15	46	12	40	50
gnuchess-6.0.0	38	1222	50	220	81	312	351
gnugo-3.8	83	2801	77	159	92	766	1493
grep-2.9	35	820	41	85	22	65	122
gzip-1.4	27	494	22	268	91	303	230
lapack-3.3.1	954	16422	294	3740	3773	8159	10351
make-3.82	34	993	67	108	53	109	257
tar-1.26	73	1712	37	218	115	253	396

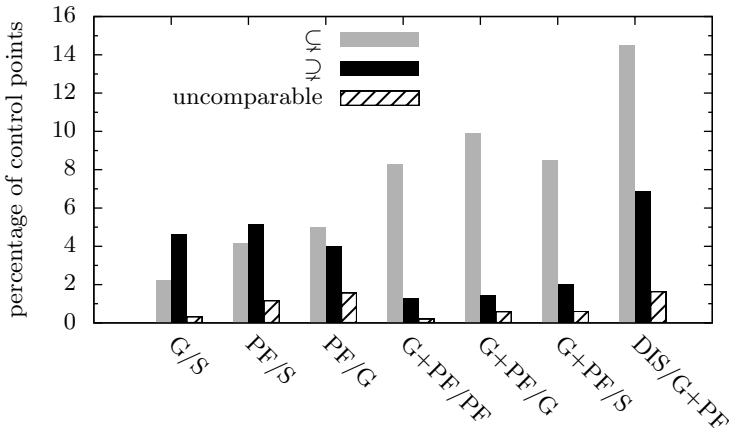


Fig. 3. Comparison of the abstract values obtained on several open-source projects. The table shows their respective number of lines of code, number of control points in P_R , and execution time on various techniques. Techniques are classical abstract interpretation (S), *Guided Static Analysis* (G), *Path-focused* technique (PF), our combined technique (G+PF), and its version with disjunctive invariants (DIS). The \subseteq bars (resp. \supseteq) gives the percentage of invariants stronger (more precise; smaller with respect to inclusion) with the left-side (resp. 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.

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. 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 (2005)
2. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library, version 0.9, <http://www.cs.unipr.it/pp1>
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 (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. Balakrishnan, G., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Refining the control structure of loops using static analysis. In: *EMSOFT*, pp. 49–58. ACM (2009)
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. Cousot, P., Cousot, R.: Abstract interpretation frameworks. *J. of Logic and Computation*, 511–547 (August 1992)
9. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ Analyzer. In: Sagiv, M. (ed.) *ESOP 2005*. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
10. 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)

11. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
12. Gawlitza, T., Monniaux, D.: Improving Strategies via SMT Solving. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 236–255. Springer, Heidelberg (2011)
13. Gonnord, L., Halbwachs, N.: Combining Widening and Acceleration in Linear Relation Analysis. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 144–160. Springer, Heidelberg (2006)
14. Gopan, D., Reps, T.W.: Guided Static Analysis. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 349–365. Springer, Heidelberg (2007)
15. Gulwani, S., Zuleger, F.: The reachability-bound problem. In: PLDI, pp. 292–304. ACM (2010)
16. Halbwachs, N.: Détermination automatique de relations linéaires vérifiées par les variables d’un programme. Ph.D. thesis, Grenoble University (1979)
17. 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 (1997)
18. Harris, W.R., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Program analysis via satisfiability modulo path programs. In: POPL, pp. 71–82. ACM (2010)
19. Henry, J.: Static Analysis by Path Focusing. Master’s thesis, Grenoble INP (2011), http://www-verimag.imag.fr/~jhenry/pdf/M2R_report.pdf
20. Jeannet, B.: Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in System Design* 23(1), 5–37 (2003)
21. Jeannet, B., Miné, A.: APRON: A Library of Numerical Abstract Domains for Static Analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009)
22. Kroening, D., Strichman, O.: *Decision procedures*. Springer (2008)
23. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO, pp. 75–86. IEEE Computer Society, Washington, DC (2004)
24. LLVM team: LLVM Language Reference Manual (2011), <http://llvm.org/docs/LangRef.html>
25. Miné, A.: The octagon abstract domain. *Higher-Order and Symbolic Computation* 19(1), 31–100 (2006)
26. Monniaux, D., Bodin, M.: Modular Abstractions of Reactive Nodes Using Disjunctive Invariants. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 19–33. Springer, Heidelberg (2011)
27. Monniaux, D., Gonnord, L.: Using Bounded Model Checking to Focus Fixpoint Iterations. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 369–385. Springer, Heidelberg (2011)
28. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
29. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. *Transactions on Programming Languages and Systems (TOPLAS)* 29(5), 26 (2007)
30. Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying Loop Invariant Generation Using Splitter Predicates. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 703–719. Springer, Heidelberg (2011)