Policy Iteration within Logico-Numerical Abstract Domains

Pascal Sotin[1], Bertrand Jeannet[1], Franck Védrine[2], and Eric Goubault[2]

[1] INRIA, {Pascal.Sotin,Bertrand.Jeannet}@inria.fr
[2] CEA-LIST LMeASI, {Frank.Vedrine,Eric.Goubault}@cea.fr

Abstract. Policy Iteration is an algorithm for the exact solving of optimization and game theory problems, formulated as equations on min max affine expressions. It has been shown that the problem of finding the least fixpoint of semantic equations on some abstract domains can be reduced to such optimization problems. This enables the use of Policy Iteration to solve such equations, instead of the traditional Kleene iteration that performs approximations to ensure convergence.

We first show in this paper that under some conditions the concept of Policy Iteration can be integrated into numerical abstract domains in a generic way. This allows to widen considerably their applicability in static analysis. We consider here the verification of programs manipulating Boolean and numerical variables, and we provide an efficient method to integrate the concept of policy in a logico-numerical abstract domain that mixes Boolean and numerical properties. Our experiments shows the benefit of our approach compared to a naive application of Policy Iteration to such programs.

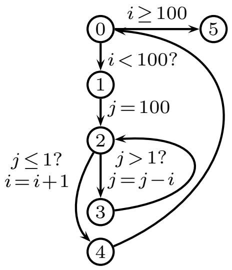
1 Introduction

Kleene Iteration. Abstract Interpretation is a framework for solving verification problems expressed by semantic equations on a (concrete) lattice. Typically, it is used to compute an overapproximation of the reachable states of a program. The computation is performed by a *Kleene iteration* which starts at the bottom of an (abstract) lattice and applies the semantic equations until no new state is reached. In order to ensure and accelerate the termination of this process, an extrapolation operator (called *widening*) is used at the cost of some additional approximation. Eventually, the result can be refined in a process called *narrowing*. The whole process is called *accelerated Kleene iteration* (pictured on Fig. 3).

Running example. Consider the program of Fig. 1(a), taken from [CGG+05]. It contains two loops and two integer variables. If the program reaches the program point ⑤, then $\mathtt{i} = 100$. However, the *accelerated Kleene iteration* will fail to infer it, because of the nested loops. In practice, performing abstract interpretation using boxes (aka. intervals) will infer the invariant $\mathtt{i} \geq 100$. The reason is that the *Kleene iteration* applies on the Control Flow Graph (CFG) of Fig. 1(b). In

```
int i=1; ⓪
while (i<100){ ①
  int j=100; ②
  while (j>1){
   j = j-i; ③
  }
  i = i+1; ④
} ⑤
end
```

(a) Original Program



(b) CFG

```
bool always,ext,inner;
int i=1;
while ((always||ext)&&i<100){
  int j=100;
  while ((always||inner)&&j>1){
   j = j-i;
  }
  i = i+1;
}
end
```

(c) Addition of Booleans

Fig. 1. Two loops running example

this graph, imprecision introduced by the widening applied for the outer loop is kept in the inner loop and prevents the narrowing from exploiting `i<100`. The problem we face here is not a weakness of the abstract domain, since the octagons or the polyhedra do not infer either this i = 100, but a weakness of the accelerated Kleene Iteration.

Policy Iteration. Introduced in [CGG+05], the use of *policy iteration* techniques for solving semantics equations with fixpoint allows to infer box-like invariants, among which the correct invariant at program point ⑤. This technique avoids the inaccuracy issues faced by the accelerated Kleene iteration. The algorithm of [CGG+05] combines an iteration on a set of policies, that defines sound variations of the semantic equations, with a linear programming solver.

The use of linear programming to solve the semantic equations is adapted for linear programs, with linear guards. It is unfortunately not adapted to a wider class of programs. For example, let consider the program of Fig. 1(c), which introduces some Boolean variables in the program of Fig. 1(a).

– Linear programming does not handle the Boolean variables.
– Existing abstract domains do handle the Boolean variables, but the accelerated Kleene iteration is inaccurate.

This article aims at taking the best of both world by performing *policy iteration on accelerated Kleene iterations.* In particular we address the question of dealing with programs having both Boolean and numerical variables (eg. Fig. 1(c)).

Contributions. We allow policy iteration on abstract semantic equations by integrating the policies for numerical abstract domains in the generic abstract domain library Apron (see Section 4). This enables the precise analysis of Fig. 1(a) in the Abstract-Interpretation-based tools Interproc. We then prove the interest of this integration by building policies for logico-numerical abstract domains on top of our numerical policies. These policies have been implemented using Mtbdds and integrated in the BddApron library (see Section 5). We could eventually perform the analysis of Fig. 1(c), for which Kleene iteration is not precise and for which policy iteration of [CGG+05] is not possible as is.

Outline. Section 2 recalls the basics of Abstract Interpretation, focusing on the abstract domains. Section 3 details the use of Kleene iteration and policy iteration for the resolution of semantic equations. Sections 4 and 5 present our contributions. Section 6 provides experiments which illustrates the questions of precision and efficiency. Section 7 will conclude and emphasize the interests of integrating the precision improvements due to policy iteration into traditional abstract interpretation frameworks.

2 Abstract Interpretation and Abstract Domains

Many static analysis problems come down to the computation of the least solution of a fixpoint equation $X = F(X), X \in C$ where C is a domain of concrete properties, and F a function derived from the semantics of the analysed program. Abstract Interpretation [CC92] provides a theoretical framework for reducing this problem to the solving of a simpler equation

$$Y = G(Y), Y \in A \tag{1}$$

in a domain A of *abstract properties.* Having performed this *static approximation*, one is left with the problem of solving Eqn. (1). The paper contributes to this problem, which is detailed in the next section.

We need however to detail first how this general method will be instantiated.

- We consider simple programs without procedures that manipulate n scalar variables taking their values in a set D, as exemplified by the programs of Figs. 1(a) and 1(c). Their state-space has the structure $S = K \times D^n$, where K is the set of nodes of the control flow graph (CFG).
- We focus on the inference of invariants. The domain of concrete properties is $C = \mathcal{P}(S) = K \to \mathcal{P}(D^n)$: an invariance property is defined by the set of possible values for variables at each node.
- The equation to be solved is $X = F(X) = I \cup \mathit{post}(X)$, where I is the set of initial states and *post* is the successor-state function. The least solution $\mathit{lfp}(F)$ of this equation is the strongest inductive invariant of the program. This equation is actually partitioned along the nodes and edges of the CFG:

$$X^k = I^k \cup \bigcup\nolimits_{(k',k)} [\![op^{(k',k)}]\!](X^{k'}) \ , \ X^k \in \mathcal{P}(D^n) \tag{2}$$

  $[\![op^{(k',k)}]\!] : \mathcal{P}(D^n) \to \mathcal{P}(D^n)$ reflects the semantics of the program instruction $op^{(k',k)}$ associated with the CFG edge (k', k). We consider here for op assignments $x := \mathit{expr}$ and tests $\mathit{bexpr}?$.
- Given an abstract domain A for $\mathcal{P}(D^n)$, abstracting Eqn. (2) in A consists in substituting $\cup$ and $[\![op]\!]$ functions in it with their abstract counterpart denoted with $\cup^\sharp$, $[\![op]\!]^\sharp$. We obtain a system

$$Y^k = I^{\sharp k} \cup^\sharp \bigcup\nolimits^{\sharp}_{(k',k)} [\![op^{(k',k)}]\!]^\sharp (Y^{k'}) \ , \ Y^k \in A \tag{3}$$

We refer to [CC92] for the conditions ensuring the soundness of the approach.

Numerical abstract domains. If the considered program manipulates only numerical variables, $D = \mathbb{Q}$, and $C = K \rightarrow \mathcal{P}(\mathbb{Q}^n)$. Many *numerical* abstract domains have been designed for approximating subsets of $\mathbb{Q}^n$:

– The *box* domain [CC76] approximates such subsets by their bounding boxes. The abstract semantics of assignments and conditionals is based on classical interval arithmetic.

– The *octagons* domain [Min06] approximates such subsets by conjunction of $\mathcal{O}(n^2)$ inequalities of the form $a_i x_i + a_j x_j \geq b$ where $a_i, a_j \in \{-1, 0, 1\}$ and the bounds b's are inferred. The abstract semantics of octagons relies on a mixture of interval arithmetic and constraint propagation.

– These two domains are generalized by the *template polyhedra* domain [SSM05] that considers conjunctions of M linear inequalities of the form $\boldsymbol{T}_m \cdot \boldsymbol{x} \geq b_m, 1 \leq m \leq M$, where the $\boldsymbol{T}_m$ are linear expressions provided by some external means and the bounds b_m are inferred. The abstract semantics is computed by linear programming.

Observe that some domains are more complex, like the *convex polyhedra* domain [HPR97] that approximates numerical subsets by convex polyhedra: it infers not only bounds, but also the (unbounded) set of linear expressions to be bounded.

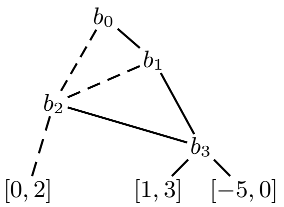
Besides the three major operations mentioned above (union, assignments and tests), other operations like existential quantification and intersection are needed for the analysis of programs with scoping rules and procedure calls. The Apron library [JM09] provides a common high-level API to such numerical domains, and defines a rich concrete semantics (including non-linear and floating-point expressions and constraints) that should be correctly abstracted by the compliant abstract domains.

The BddApron logico-numerical abstract domain. The Apron concrete semantics and the abstract domains provided with it do not provide the adequate operations for programs that manipulate also Boolean and enumerated variables, which may contain instructions like

`x := if b and x<=5 then x+1 else 0` or `b := b and x<=3`

In this case $D = \mathbb{B} \uplus \mathbb{Q}$ and $\mathcal{P}(D^n) \simeq \mathcal{P}(\mathbb{B}^p \times \mathbb{Q}^q)$. A naive solution is to eliminate Boolean variables by encoding them in the control, so as to obtain a purely numerical program. However this solution (i) is neither efficient – the enumeration of Boolean valuations induces an exponential blow-up, (ii) nor it provides a high-level view on invariants and their manipulation.

The BddApron library [Jea] addresses issue (ii) by offering support for expressions and constraints that freely combine Boolean and numerical subexpressions and by leveraging any Apron-compliant numerical abstract domain to a *logico-numerical* abstract domain for such a concrete semantics. Given a numerical abstract domain A_0 for $\mathcal{P}(\mathbb{Q}^q)$, it abstracts concrete properties in $\mathcal{P}(\mathbb{B}^p \times \mathbb{Q}^q) \simeq \mathbb{B}^p \rightarrow \mathcal{P}(\mathbb{Q}^q)$ with functions in $\mathbb{B}^p \rightarrow A_0$. The efficiency issue (i) is addressed by representing functions $f : \mathbb{B}^p \rightarrow A_0$ with Mtbdds [Bry86], see Fig. 2. This representation does not improve the worst-case complexity in $\mathcal{O}(2^p)$, but the complexity of the representation and of the operations becomes



Represents the function
$(\neg b_0 \vee \neg b_1) \wedge \neg b_2 \mapsto [0, 2]$
$(\neg b_0 \wedge b_2 \vee b_0 \wedge b_1)$
$\wedge \neg b_3 \mapsto [1, 3]$
$(\neg b_0 \wedge b_2 \vee b_0 \wedge b_1)$
$\wedge b_3 \mapsto [-5, 0]$
of signature $\mathbb{B}^4 \to \mathbb{I}$

Fig. 2. Example of MTBDD

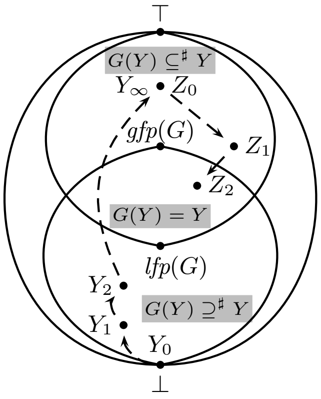


Fig. 3. Kleene iteration with widening and narrowing

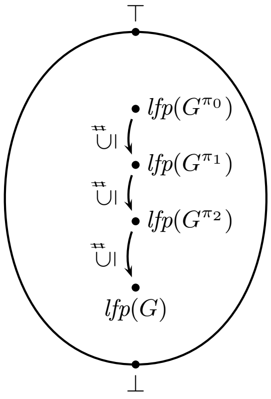


Fig. 4. Policy iteration

a function of the number of nodes of the BDDs/MTBDDs rather than a function of the number of (reachable) Boolean valuations. As in many applications the average number of nodes of MTBDDs is much smaller than the worst case 2^p, the practical complexity is significantly improved.

The contribution of this paper is to show how policy iteration solving techniques, which are described in the next section and currently apply to equations on numerical properties, can be *efficiently leveraged* to equations on logico-numerical properties by integrating them in the abstract domain in a generic way.

3 Abstract Equation Solving and Policy Iteration

The traditional way to solve the abstract semantic equation $Y = G(Y), Y \in A$ (*e.g.,* Eqn. (1)) is *Kleene iteration with widening and narrowing*. This consists in computing successively (*c.f.* Fig. 3)

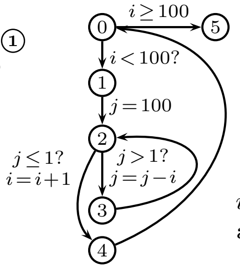
- the ascending sequence $Y_0 = \perp$, $Y_{n+1} = Y_n \nabla G(Y_n)$, which converges in a finite number of steps to a post-fixpoint Y_∞;
- the descending sequence $Z_0 = Y_\infty$, $Z_{n+1} = G(Z_n)$, up to some rank N.

$\nabla : A \times A \to A$ is a *widening* operator that ensures convergence at the cost of additional *dynamic* approximations. The problem is that such approximations are often too strong, and that the descending sequence often fails to recover useful information, as discussed in the introduction. This is why this paper focuses on an alternative resolution method.

Policy iteration is an algorithm that has been developed originally in control and game theory. It has been introduced by Howard [How60] and then extended by Hoffman and Karp [HK66] for stochastic games. It basically finds the value of a game, which is the unique fixpoint of the Shapley operator [Sha53], which is the min of a max of certain affine functions.

```
int i=1; ⓪
while (i<100){ ①
  int j=100; ②
  while (j>1){
   j = j-i; ③
  }
  i = i+1; ④
} ⑤
end
```

(a) Program



(b) CFG

$$
\begin{array}{rl}
i_0 &= [1,1] \cup^\sharp i_4 \\
i_1 &= [\![i < 100?]\!]^\sharp (i_0) \\
(i_2, j_2) &= (i_1, [100, 100]) \cup^\sharp (i_3, j_3) \\
(i_3, j_3) &= [\![j = j - i]\!]^\sharp \circ [\![j > 1?]\!]^\sharp (i_2, j_2) \\
(i_4, j_4) &= [\![i = i+1]\!]^\sharp \circ [\![j \leq 1?]\!]^\sharp (i_2, j_2) \\
i_5 &= [\![i \geq 100?]\!]^\sharp (i_0)
\end{array}
$$

$i_k = [i_k^-, i_k^+]$ and $j_k = [j_k^-, j_k^+]$ are the intervals associated with var. i and j at CFG node ⓚ

(c) Abstract Box Semantics

$$
\begin{array}{ll}
i_0^- = \min(1, i_4^-) & i_0^+ = \max(1, i_4^+) \\
i_1^- = i_0^- & i_1^+ = \underline{\min}(99, i_0^+) \\
i_2^- = \min(i_1^-, i_3^-) & i_2^+ = \max(i_1^+, i_3^+) \\
j_2^- = \min(100, j_3^-) & j_2^+ = \max(100, j_3^+) \\
i_3^- = i_2^- & i_3^+ = i_2^+ \\
j_3^- = \underline{\max}(2, j_2^-) - i_2^+ & j_3^+ = j_2^+ - i_2^- \\
i_4^- = i_2^- + 1 & i_4^+ = i_2^+ + 1 \\
j_4^- = j_2^- & j_4^+ = \underline{\min}(1, j_2^+) \\
i_5^- = \underline{\max}(100, i_0^-) & i_5^+ = i_0^+
\end{array}
$$

($\underline{\min}$ and $\underline{\max}$ are min max policies)

(d) Equivalent equations on bounds

$$
\begin{array}{rl}
-i_0^- &= \max(-1, -i_4^-) \\
-i_1^- &= -i_0^- \\
-i_2^- &= \max(-i_1^-, -i_3^-) \\
-j_2^- &= \max(-100, -j_3^-) \\
-i_3^- &= -i_2^- \\
-j_3^- &= \underline{\min}(-2, -j_2^-) + i_2^- \\
-i_4^- &= -i_2^- - 1 \\
-j_4^- &= -j_2^- \\
-i_5^- &= \underline{\min}(-100, -i_0^-)
\end{array}
$$

(e) Normalizing equations on inf bounds

Fig. 5. Abstract Interpretation and Game Theory views of semantic equations on boxes

Abstract semantic equations as min-max affine equations. As observed in [CGG+05], the abstract box semantics of programs with linear assignments and conditionals can be formulated as equations on lower and upper bounds, in which each bound is the min of a max of affine functions. Fig. 5 illustrates this point. Fig. 5(c) instantiates Eqn. (3) on the program of Fig. 5(a). Fig. 5(d) reformulates this as equations on bounds. Selecting the least solution in Fig. 5(c) is equivalent to maximizing lower and minimizing upper bounds in Fig. 5(d). In order to regularize this problem, we replace lower bounds of intervals with upper bounds by negating them, see Fig. 5(e), so as to manipulate only upper bounds subject to minimization. Such a formulation can be viewed a deterministic game problem between a min-player and a max-player. Several plays are possible, but we are interested in the play that minimizes the bounds. Min policy iteration provides a solution for this problem, by finding the optimal strategy (*i.e. policy*) of the min player.

Policy and policy iteration. In the context of an equation $Y = G(Y)$ where Y is a vector of upper bounds and G a min of max of affine functions, a *(min) policy* π is a choice of one argument per min in G, which results in a simpler function $G^\pi \sqsupseteq^\sharp G$ which is the max of affine functions. By observing that for any fixpoint of G and any min operator in G, the min will be reached by at least one argument, one deduces that the least fixpoint of G is also the least fixpoint of some G^π.

The policy iteration algorithm, illustrated by Fig. 4, works by

1. choosing an initial policy π_0;
2. at each step k, computing the least solution $Y_i = \mathit{lfp}(G^{\pi_i})$ of $Y = G^{\pi_i}(Y)$;
3. if Y_i is a solution of $Y = G(Y)$, the algorithm terminates, otherwise the *policy improvement step* consists in choosing a new policy π_{i+1} such that $\mathit{lfp}(G^{\pi_{i+1}}) \sqsubseteq^\sharp \mathit{lfp}(G^{\pi_i})$, and to go back to step 2.

   How is it done ? As $G^{\pi_i} \sqsupseteq^\sharp G$ and $Y \neq G(Y)$, $Y_i = G^{\pi_i}(Y_i) \sqsupsetneq^\sharp G(Y_i)$. Therefore, for some p^{nth} component of the vector Y_i, we have
   $$G(Y_i)^{(p)} = \min(e_1, \dots, e_n) < Y_i^{(p)} = (G^{\pi_i}(Y_i))^{(p)} = e_j$$
   where j results from the choice performed by the policy π_i, and $e_1, \dots, e_n$ are the values of the max expressions evaluated on Y_i. The principle is to replace in π_{i+1} the choice j by a choice j' such that $e_{j'} = \min(e_1, \dots, e_n)$. This ensures that $G^{\pi_{i+1}}(Y_i) \sqsubsetneq^\sharp Y_i = \mathit{lfp}(G^{\pi_i})$, hence $\mathit{lfp}(G^{\pi_{i+1}}) \sqsubsetneq^\sharp \mathit{lfp}(G^{\pi_i})$.

It is shown in [CGG$^+$05] that for boxes, this method will terminate on a fixpoint of G, which is guaranteed to be the least fixpoint (when taking care of degenerate cases) when G is not expansive for the sup norm. Some improvements of the original method of [CGG$^+$05] have been made for dealing with degenerate cases in an efficient manner in [AGG08]. Extensions of the method to deal with the zone, octagon, linear and quadratic templates are discussed in [GGTZ07,AGG10].

Policy iteration can also be seen as a Newton method for solving a system of min-max equation $Y = G(Y)$. Any of the expressions under the min operator can indeed be seen as a possible differential/linearization of G. A policy is the choice of such a differential, and solving $Y = G^\pi(Y)$ is akin to solving the linearization of G in one step in the classical Newton method. This view was exploited for *max* policy iteration in [GS07b,GS07a,GS10].

Two methods for solving $Y = G^\pi(Y)$. Once a (min) policy π is applied, one have to compute the least solution of a simpler equation $Y = G^\pi(Y)$ where G is the max of affine functions. This can be done either by linear programming as in [GGTZ07], or by standard Kleene iteration as in [CGG$^+$05].

1. Linear programming always computes the least solution, but presents some shortcomings:
   (a) It requires to write down the full equation system on bounds (whereas Kleene iteration works in practice by incremental exploration);
   (b) It does not allow to see the abstract domain (boxes, octagons, ...) and a policy linked to it as an abstract datatype (ADT).
   (c) If the program contains non-linear expressions, these must be linearized statically *before* the analysis (thus when no information is available...)
2. Kleene iteration with widening does not offer the guarantee of delivering the least solution, thus theoretical guarantees about policy improvement does not apply any more. However it exhibits better behaviour w.r.t. the points mentioned above:

(a)(b) It integrates well in existing static analysers (such as INTERPROC, [JM09,JAL]) that manipulates abstract properties as abstract datatypes through normalized APIs (such as the APRON and the BDDAPRON APIs mentioned in Section 2).

(c) Linearization of non-linear expressions can be done dynamically as in [Min02], using the (under)approximations provided by the current Kleene iteration step.

One might object that as this technique still resorts to widening to ensure convergence, it should not improve on traditional Kleene iteration (without policies). The point is actually that here Kleene iteration is applied to simpler equations, with fewer dependency cycles (hence less widening points) and on which the descending sequences is likely to be more effective. The experiments in [CGG$^+$05] and Section 6 confirms this conjecture.

For example, on Fig. 5, if one chooses the left policy for all min equations, like $i_1^+ = 99$ for the policy $i_1^+ = \min(99, i_0^+)$, the Kleene iteration solves the simpler equations in one iteration and finds $i_1 = [100, 100]$ to be compared to $i_1 = [100, +\infty]$ obtained by the global box iterations without policies[3].

In the next section, we show how the concept of policy can be integrated in an abstract domain and can be viewed as an ADT. This allows in Section 5 to leverage the use of policy iterations in logico-numerical domains.

4 Integrating Policies in Numerical Abstract Domains

Integrating policies in an abstract domain as described in Section 2 means in practice to abstract the process of translating the equations of Fig. 5(c) to the equations of Fig. 5(d) (in the case of the box abstract domain) and to "instrument" the former equations with policies.

Instrumenting abstract operations with policies. The original semantic equations are made of the three operators described in Section 2: (i) $\cup^\sharp$, (ii) $[\![bexpr?]\!]^\sharp$, and (iii) $[\![x := expr]\!]^\sharp$. For all of the template-based numerical abstract domains for which policies have been used, min operators are introduced only by tests (ii) and assignments (iii). Hence only those two latter operations needs to be equipped with a policy. We thus introduced in the APRON API two new generic functions:

$$\begin{array}{rl} \mathsf{meet\_cond\_apply\_policy}_0 : & P_0 \times A_0 \times \quad \mathit{Cond}_0 \quad \to A_0 \\ \mathsf{assign\_var\_apply\_policy}_0 : & P_0 \times A_0 \times \mathit{Var} \times \mathit{Expr}_0 \to A_0 \end{array} \qquad (4)$$

where P_0 denotes the set of policies, A_0 the numerical abstract domain, Expr_0 the set of (linear) numerical expressions, and Cond_0 the set of Boolean formula on (linear) numerical constraints under disjunctive normal form (DNF).

The exact structure of policies depends on the considered abstract domain. We illustrate the case of the box abstract domain. In this domain, min expressions will be always decomposed into min expressions with two operands:

[3] Appendix A unrolls in parallel the Kleene iteration and the best policy.

$\min(e_1, e_2)$. Therefore, the domain of a *bound policy* is $\{l, r\}$, which stands for **l**eft and **r**ight: (l) if $\pi = l$, $\min^{\pi}(e_1, e_2) = e_1$, (r) if $\pi = r$, $\min^{\pi}(e_1, e_2) = e_2$. Consider now the intersection of an abstract property $a = \prod_{k=1}^{n} I_k$ with a single numerical constraint $c = \sum\limits_{k' \in K'} \alpha_{k'} x_{k'} - \sum\limits_{k'' \in K''} \alpha_{k''} x_{k''} + \beta \geq 0$ with $\alpha_{k'}, \alpha_{k''} > 0$ and $K' \cap K'' = \emptyset$. We want to express $a' = \mathsf{meet\_cond\_apply\_policy}_0(\pi, a, c)$. The constraint c can be rewritten as

$$\begin{array}{ll} x_k \geq \frac{1}{\alpha_k}\Big(\sum\limits_{k' \in K' \setminus \{k\}} -\alpha_{k'} x_{k'} + \sum\limits_{k'' \in K''} \alpha_{k''} x_{k''} + \beta \Big) & \text{if } k \in K' \\ x_k \leq \frac{1}{\alpha_k}\Big(\sum\limits_{k' \in K'} \alpha_{k'} x_{k'} - \sum\limits_{k'' \in K'' \setminus \{k\}} \alpha_{k''} x_{k''} + \beta \Big) & \text{if } k \in K'' \end{array}$$

Hence $a' = \prod_{k=1}^{n} I'_k$ can be expressed as:

$$-(I'_k)^- = \begin{cases} \min^{\pi_{k,-}}\Big(-I_k^-, \frac{1}{\alpha_k}\Big(- \sum\limits_{k' \in K' \setminus \{k\}} \alpha_{k'} I_{k'}^- + \sum\limits_{k'' \in K''} \alpha_{k''} I_{k''}^+ + \beta \Big)\Big) & \text{if } k \in K' \\ -I_k^- & \text{otherwise} \end{cases}$$

$$(I'_k)^+ = \begin{cases} \min^{\pi_{k,+}}\Big(I_k^+, \frac{1}{\alpha_k}\Big(\sum\limits_{k' \in K'} \alpha_{k'} I_{k'}^+ - \sum\limits_{k'' \in K'' \setminus \{k\}} \alpha_{k''} I_{k''}^- + \beta \Big)\Big) & \text{if } k \in K'' \\ I_k^+ & \text{otherwise} \end{cases}$$

In practice, we associate a bound policy $\pi_{k,+/-}$ to each interval bounds, hence $\pi \in \{l, r\}^{2q}$ for the intersection with a single linear inequality in q dimensions. Equalities are handled as the conjunction of two inequalities. This "instrumentation" with policies is generalized to conjunctions of m linear inequalities and equalities, which results in a policy in $\{l, r\}^{2qm}$. The meet of a with a general Boolean formula under DNF form $\bigvee_{i=1}^{p} \bigwedge_j c_{i,j}$ is handled as the disjunction of the meet of a with the p conjuncts $\bigwedge_j c_{i,j}$.

Assignments do not imply min operators in the box abstract domains. On octagons an assignment like $x_1 = 2x_2 + 4$ is performed by introducing a primed variable x'_1, intersecting the octagon with $x'_1 = 2x_2 + 4$ (*which implies min operators*), eliminating x_1 and renaming x'_1 in x_1. Still, ultimately only the $\mathsf{meet\_cond}$ operation needs to be equipped with a policy. It is however not the case for more general linear templates.

Improving policies. Remind from Section 3 that given a solution $Y = G^{\pi}(Y)$, we need to improve the policy π if $G(Y) \subsetneq^{\sharp} Y$. We thus introduce in the API two new generic functions

$$\begin{array}{rl} \mathsf{meet\_cond\_improve\_policy}_0 : P_0 \times A_0 \times & \mathit{Cond}_0 \quad \to P_0 \\ \mathsf{assign\_var\_improve\_policy}_0 : P_0 \times A_0 \times & \mathit{Var} \times \mathit{Expr}_0 \to P_0 \end{array} \tag{5}$$

$\mathsf{meet\_cond\_improve\_policy}(\pi, a, c)$ proceeds as follows ($\mathsf{assign\_var\_improve\_policy}_0$ proceeds exactly in the same way).

- it computes $a' = \mathsf{meet\_cond}_0(a, c)$ and $a'' = \mathsf{meet\_cond\_apply\_policy}_0(\pi, a, c)$;
- if $a' = a''$, it returns π; otherwise, it chooses a new policy π' such that $a' = \mathsf{meet\_cond\_apply\_policy}_0(\pi', a, c)$, following the principle explained in Section 3, and it returns it.

Integration in the policy iteration process. Once abstract operations are instrumented with policies, one parametrizes Eqn. (3) by associating to each operation $[\![op^{(k',k)}]\!]^\sharp$ a policy $\pi^{(k',k)}$:

$$Y^k = I^{\sharp k} \cup^\sharp \bigcup\nolimits^\sharp_{(k',k)} \mathsf{op\_apply}^{(k',k)}(\pi^{(k',k)}, Y^{k'}, args\ldots) \tag{6}$$

We apply the process described in Section 3. We fix an initial global policy π_0, and at each policy iteration step i,

1. We solve Eqn. (6) with $\pi = \pi_i$ using Kleene iteration with widening and narrowing; we obtain a solution Y_i.
2. We compute the new policy with $\pi_{i+1}^{(k',k)} = \mathsf{op\_improve}(\pi_i^{(k',k)}, Y_i^{(k')}, args\ldots)$. If $\pi_{i+1} \neq \pi_i$, we iterate the process, otherwise we have a solution.

Implementation. Augmenting the APRON API with the 4 functions introduced by Eqns. (4)-(5) allowed us to integrate nicely policy iteration in the INTERPROC interprocedural analyser, based on the APRON numerical abstract domain libraries and the FIXPOINT generic equation solver [Jea10]. Currently, we have implemented these functions only for the box abstract domain. In the static analyser, we needed to add about 100 OCaml LOC to take care of the policy iteration process (creating policies, updating them and testing convergence). Once a policy π is fixed, we reuse the existing code for solving the equation $Y = G^\pi(Y)$.

As INTERPROC also addresses recursive programs, two additional abstract operations appear in the semantic equations: (i) procedure call, which involves projection and variable renaming, hence no policy; (ii) procedure returns, which involves the meet of two abstract values. We did not yet instrument the meet operation, but there is no theoretical problem to do it. Moreover, as we solve $Y = G^\pi(Y)$ by Kleene iteration, we can deal with more complex functions G^π than if we were tight to problems expressed as linear programs.

5 Policy for Logico-Numerical Abstract Domain

We showed in Section 4 how the concept of policy can be integrated into a numerical abstract domain in a generic way. The practical advantage was the ability to add the *boxpolicy* domain to the APRON library, and ultimately to the INTERPROC analyser, and to benefit for (almost) free from all the techniques it implements (*e.g.,* non-linear arithmetic and interprocedural analysis). In this section we show that this integration can be pushed further to the BDDAPRON logico-numerical abstract domain, which acts as a functor on top of an APRON domain. Moreover, we show that this can be implemented efficiently with MTBDDS.

BddApron abstract operation. As explained in Section 2, the BDDAPRON library proposes to abstract logico-numerical properties in $\mathcal{P}(\mathbb{B}^p \times \mathbb{Q}^q)$ by functions in $A = \mathbb{B}^p \to A_0$. Extending the conditional and assignment operations from A_0 to A is easy under the following conditions:

$$\begin{aligned}
\mathsf{meet\_cond} &: A \times \quad Cond \quad \to A \\
\mathsf{assign\_var} &: A \times \; NVar \times Expr \; \to A \\
\mathsf{assign\_bvar} &: A \times BVar \times BExpr \to A \\
\mathsf{meet\_cond}(f,c) &= \lambda \boldsymbol{b} \,.\, \mathsf{meet\_cond}_0\big(f(\boldsymbol{b}), c(\boldsymbol{b})\big) \\
\mathsf{assign\_var}(f,x_k,e) &= \lambda \boldsymbol{b} \,.\, \mathsf{assign\_var}_0\big(f(\boldsymbol{b}), x_k, e(\boldsymbol{b})\big) \\
\mathsf{assign\_bvar}(f,b_k,\varphi) &= \lambda \boldsymbol{b} \,.\, \begin{cases} \quad\;\, (\text{if } b_k \Leftrightarrow \varphi^+(\boldsymbol{b}) \text{ then } f^+(\boldsymbol{b}) \text{ else } \bot_0) \\ \cup_0^\sharp \; (\text{if } b_k \Leftrightarrow \varphi^-(\boldsymbol{b}) \text{ then } f^-(\boldsymbol{b}) \text{ else } \bot_0) \end{cases} \\
& \quad \text{where } f = ite(b_k, f^+, f^-) \text{ and } \phi = ite(b_k, \phi^+, \phi^-) \\
& \quad\quad \text{are decomposed into their cofactors w.r.t. } b_k
\end{aligned}$$

Fig. 6. BddApron abstract operations

- Conditions in tests are put under the form $Cond = \mathbb{B}^p \to Cond_0$.
- Assigned expressions are
  - either numerical expressions in $Expr = \mathbb{B}^p \to Expr_0$;
  - or purely Boolean expressions in $BExpr = \mathbb{B}^p \to \mathbb{B}$.

  In other words, they do not involve conditions on numerical variables. Examples are `b0 = (b1 or (b2 and not b3))`, `x0 = (if b1 then x1+1 else x2-1)`.

Under these assumptions where the conditions and expressions are pointwise extensions of the conditions and expressions considered in A_0, tests and assignments in A can be defined as in Fig. 6

Notice that "forbidden" assignements like `x0 = (if x0>10 then 0 else x0+1)` or `b0 = (x0>=0)` can be emulated by replacing conditional expressions with conditional assignments. The BddApron library actually handles them directly, but this requires more complex algorithms that makes difficult their instrumentation with policies discussed below.

Boolean extension of numerical operations with policies. Observe the $\mathsf{meet\_cond}$ operation in Fig. 6: it applies pointwise the $\mathsf{meet\_cond}_0$ operation to $f(\boldsymbol{b})$ and $c(\boldsymbol{b})$ for every $\boldsymbol{b} \in \mathbb{B}^p$. If we want to parameterize it with a policy, we need one policy $\pi(\boldsymbol{b}) \in P_0$ for each $\boldsymbol{b} \in \mathbb{B}^p$. If we have such a *logico-numerical policy* $\pi : \mathbb{B}^p \to P_0$, we apply $\mathsf{meet\_cond\_apply\_policy}_0$ pointwise to $\pi(\boldsymbol{b})$, $f(\boldsymbol{b})$ and $c(\boldsymbol{b})$ for each $\boldsymbol{b} \in \mathbb{B}^p$. We get the following definition.

Definition 1 (Logico-numerical policy). *If P_0 denotes the set of policies associated with the numerical abstract domain A_0, the set of policies associated with the logico-numerical domain $A = \mathbb{B}^p \to A_0$ is* $\boxed{P = \mathbb{B}^p \to P_0}$.

The $\mathsf{op\_apply\_policy}$ and $\mathsf{op\_improve\_policy}$ operations in A are defined in Fig. 7 by extending pointwise the corresponding operations in A_0. As the operation $\mathsf{assign\_bvar}$ involves only the numerical operation $\cup_0^\sharp$, it is not need a policy.

We have set exactly the same framework than the one of Section 4. We can thus analyse logico-numerical programs with the BddApron extension of any numerical domain equipped with policies (like the box domain). In this new

$$
\begin{array}{rl}
\mathsf{meet\_cond\_apply\_policy}: & P \times A \times \quad \mathit{Cond} \quad \to A \\
\mathsf{meet\_cond\_improve\_policy}: & P \times A \times \quad \mathit{Cond} \quad \to P \\
\mathsf{assign\_var\_apply\_policy}: & P \times A \times \mathit{Var} \times \mathit{Expr} \to A \\
\mathsf{assign\_var\_improve\_policy}: & P \times A \times \mathit{Var} \times \mathit{Expr} \to P
\end{array}
$$

$$
\begin{aligned}
\mathsf{meet\_cond\_apply\_policy}(\pi, f, c) &= \lambda \boldsymbol{b}\,.\, \mathsf{meet\_cond\_apply\_policy}_0\big(\pi(\boldsymbol{b}), f(\boldsymbol{b}), c(\boldsymbol{b})\big) \\
\mathsf{meet\_cond\_improve\_policy}(\pi, f, c) &= \lambda \boldsymbol{b}\,.\, \mathsf{meet\_cond\_improve\_policy}_0\big(\pi(\boldsymbol{b}), f(\boldsymbol{b}), c(\boldsymbol{b})\big) \\
\mathsf{assign\_var\_apply\_policy}(\pi, f, x_k, e) &= \lambda \boldsymbol{b}\,.\, \mathsf{assign\_var\_apply\_policy}_0\big(\pi(\boldsymbol{b}), f(\boldsymbol{b}), x_k, e(\boldsymbol{b})\big) \\
\mathsf{assign\_var\_improve\_policy}(\pi, f, x_k, e) &= \lambda \boldsymbol{b}\,.\, \mathsf{assign\_var\_improve\_policy}_0\big(\pi(\boldsymbol{b}), f(\boldsymbol{b}), x_k, e(\boldsymbol{b})\big)
\end{aligned}
$$

Fig. 7. Parametrization of logico-numerical operations with policies

context, the solution Y_i of $Y = G^{\pi_i}(Y)$ computed by Kleene iteration actually provides two kind of informations: the set of reachable Boolean valuations at node, and the numerical invariants associated with each of them.

Implementation with Mtbdds. Our operations involve functions of signature $\mathbb{B}^p \to T$. If they are represented with a tabulated representation, the complexity of abstract operations is in $\mathcal{O}(2^p)$. In particular we need 2^p numerical policies in P_0 at each edge of the program CFG.

The solution is to reuse the principle behind the BddApron library, which is to represent functions of signature $\mathbb{B}^p \to T$ with Mtbdds [Bry86]. As mentioned in Section 2, the complexity of an operation defined as

$$
\begin{array}{rll}
\mathit{op} : (\mathbb{B}^p \to T_1) \times (\mathbb{B}^p \to T_2) & \to & (\mathbb{B}^p \to T) \\
(f_1, f_2) & \mapsto & \mathit{op}(f_1, f_2) = \lambda \boldsymbol{b}\,.\, \mathit{op}_0(f_1(\boldsymbol{b}), f_2(\boldsymbol{b})) \\
& & \text{with } \mathit{op}_0 : T_1 \times T_2 \to T
\end{array}
\tag{7}
$$

is $\mathcal{O}(2^p)$ with a tabulated representation of f_1 and f_2, and $\mathcal{O}(|f_1| \cdot |f_2|)$ with a Mtbdd representation of f_1 and f_2 with $|f_1|$ and $|f_2|$ nodes. In the latter case the function op is implemented by a parallel, recursive descent of the Mtbdds f_1 and f_2, using memoization techniques to avoid exploring already explored pairs of subgraphs. As the functions of Fig. 7 follow the pattern of Eqn. (7), they benefit from such techniques.

The condition on a set T for representing functions in $\mathbb{B}^p \to T$ with Mtbdds is the ability (i) to test the equality of two elements in T, (ii) and to have a reasonably efficient hash function. In the case of the box domain, policies are elements of sets of the form $\{l, r\}^N$, as discussed in Section 4, and meet these requirements. It is also the case for policies for the octagon domain [GGTZ07].

Concerning the initial policy, our (naive) tactic is to associate to each operation op of the CFG a constant policy $\pi_0^{(k',k)} = \lambda \boldsymbol{b}\,.\, p_0 \in P_0$. Later on, the Mtbdd size of policies may vary during the policy iteration process. This depends on the number of distinct numerical policies associated to Boolean valuations.

6 Experiments

This section presents experimental results showing that policy iteration on logico-numerical abstract domains, as presented in Section 2, allows precise and tractable

Program	Nesting	$\#\mathbb{B}$	$\#K$	Boxes only	Boxes+policies		
					No sharing	Full sharing	Prec.
test1'	1	2	4	8ms	17ms	15ms (2 it.)	$=$
test2'	1	3	5	18ms	42ms	34ms (2 it.)	$=$
test3'	1	2	4	8ms	15ms	13ms (1 it.)	$=$
test4'	1	10	12	226ms	25 300ms	480ms (3 it.)	$=$
test5'	2	4	6	23ms	79ms	47ms (2 it.)	$>$
test6'	2	6	8	44ms	520ms	124ms (3 it.)	$>$
test7'	2	6	8	40ms	310ms	81ms (2 it.)	$>$
test8'	3	6	8	60ms	280ms	113ms (2 it.)	$=$
test9'	3	6	8	58ms	360ms	116ms (2 it.)	$>$

Table 1. Experiments with modified examples of [CGG+05]

Program	Threads, $\#\mathbb{B}$, $\#\mathbb{Q}$, $\#K$	Boxes only	Boxes+policies			
			No sharing	Full sharing	Disting.	Prec.
BlueTooth	$2T, 5\mathbb{B}, 3\mathbb{Q}, 87K$	0.21s	0.99s	0.84s (3 it.)	17%	$=$
Preemptive	$2T, 9\mathbb{B}, 1\mathbb{Q}, 352K$	0.83s	18.64s	1.37s (1 it.)	0.7%	$=$
Barrier	$2T, 5\mathbb{B}, 2\mathbb{Q}, 95K$	0.79s	3.05s	1.96s (2 it.)	9.5%	$>$
Loop2TML	$2T, 1\mathbb{B}, 6\mathbb{Q}, 37K$	0.10s	0.22s	0.21s (2 it.)	70%	$>$

Table 2. Experiments with concurrent programs.

analysis of programs involving Boolean variables, numerical variables and even concurrency. The experiments were performed with the ConcurInterproc analyser, using BddApron and logico-numerical policies.

Analysis of the running example. We perform the analysis of the programs shown on Figures 1(a) and 1(c). For these two programs, the analysis with boxes (only) does not infer the most precise bounds for `i` and `j` while the analysis with boxes and policies does. The use of policy iteration have little impact on the analysis times. Thanks to the Mtbdds, the analysis times for the program of Fig. 1(c) is of the same order of magnitude than the ones of Fig. 1(a), in spite of the eight possible boolean valuations to consider.

Note also that these exact bounds found by boxes with policies cannot be inferred by expansive abstract domains like the polyhedra or the octagons.

Examples from [CGG+05] plus Booleans. We modify the programs experimented in [CGG+05] by introducing in a systematic way Boolean variables in order to demonstrate that:

1. Policy Iteration on boxes is more precise than boxes only.
2. Analysis time does not increase as fast as the number of boolean valuations.

We added a Boolean variable for each loop, each conditional and each variable modification. These Boolean variables are then used as additional condition to enter the loop, enter the `then` branch and perform the modification. For example, it introduces the uninitialized Boolean variables `a` and `b` in the following program:

```
while (x<100)         ⟶        while (a && x<100)
  x=x+1;                         if (b) x=x+1;
```

The results are shown in Table 1. The column *program* gives the name of the original program with an additional **'** to recall the transformation. The column *nesting* gives the maximum nesting depth of the loops. The columns $\#\mathbb{B}$ and $\#K$ count respectively the number of Boolean variables introduced and the number of control points. The results obtained by our approach are shown in the column *boxes+policies, full sharing* and are to be compared with the ones without policies, taking into account whether the box abstract domain reach the same precision as policies (=) or not (>).

We also experimented the loss of efficiency that could be endured if we do not share the policies. The column *no sharing* indicates the analysis time when we take one policy per Boolean valuation instead of a Mtbdd of policies.

All the analyses using policy iteration discover the *best invariant* one could hope for boxes. The symbols > indicate cases where traditional boxes cannot infer this optimal invariant. The experiments show that boxes with policy iteration timings tends to be proportional to the timings using the classical BddApron boxes multiplied by the number of iteration. The idea of applying the method of [CGG$^+$05] using one policy per boolean valuation does not scale. For example, we need to consider one thousand policies per meet operation for `test4'`.

Analysis of concurrent programs. Table 2 shows the results of experiments involving concurrent programs performing synchronisation using shared Boolean variables. The columns have to be interpreted like the ones of Tab. 1, with an additional column *disting.* containing the percentage of policies that truly need to be distinguished. Note that procedures have been inlined, and that the commutation between threads creates large control flow graphs with many cycles.

The results obtained by policy iterations can be far more precise than the ones obtained without, as it is the case for the program `Barrier` (which explains the increase of the analysis cost). The timings confirm that when both analysis are equally precise, our implementation is slower by a factor close to the number of policy explored. The experiments we have performed also showed that the iterations tend to be faster as the policies get improved.

7 Conclusion

We first showed in this paper how to integrate in a generic way the concept of policy and policy iteration into a numerical abstract domain. This is done at the cost of giving up with the ability to solve *exactly* the equation $Y = G^{\pi}(Y)$ parametrized with the policy π using linear programming[4]. However we believe that this shortcoming is largely counter-balanced by the gains, which are

(i) the easy integration in existing static analysis tool ([Jea10]);
(ii) the ability to build more complex abstract domain on top of such policy-equipped numerical domains and to address programs with other datatypes.

[4] which is possible any way only when the program does not contain non-linear arithmetic operation.

We demonstrated point (i) by equipping the box domain implemented in the Apron library with policies, and by integrating it in the Interproc tool. Our major contribution is however the demonstration of point (ii) in the case of programs manipulating Boolean and numerical variables. Instead of assigning a numerical policy to each Boolean valuation, we showed that we can use Mtbdds techniques to assign a single policy to a (potentially large) set of Boolean valuations. This efficient representation *logico-numerical policy* was integrated in the BddApron library.

Our experiments illustrated two points. They first showed that this later technique improves in a spectacular way the efficiency of policies, compared to their naive application, even for simple programs with a dozen of Boolean variables. They also showed that despite the theoretical shortcoming of our approach mentioned above w.r.t. precision, in practice our combination of policy and Kleene iteration delivers more precise results than the traditional approach that relies only on Kleene iteration.

A first perspective of this work is the use of policy iteration in complex abstract domains like the one proposed in [CR08] for dynamically allocated data-structure, which is parametrized by a numerical abstract domain. Our approach enables the use of policies in this context, whereas the traditional approach based on translation to min-max equations as in Fig. 5(d) is totally infeasible.

Another perspective would be to investigate the use of *max policy iteration* in a similar way as we did in this paper for min policy iteration. Max policy methods were put forward by Gawlitza and Seidl [GS07b,GS07a,GS10] for the same abstract domains: instead of selecting one argument of the min operators, it selects one argument of the max operators. Max and min policy iteration offer different advantages. Unlike min policy methods that over-approximate $\mathit{lfp}(G)$ until eventually reaching a fixpoint of G, max policy methods under-approximates $\mathit{lfp}(G)$; therefore they cannot be stopped before convergence, but they are guaranteed to reach $\mathit{lfp}(G)$ (and not just a fixpoint of G) for a larger class of programs.

References

[AGG08] A. Adjé, S. Gaubert, and E. Goubault. Computing the smallest fixpoint of nonexpansive mappings arising in game theory and static analysis of programs, July 2008.

[AGG10] A. Adjé, S. Gaubert, and E. Goubault. Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. In *European Symposium on Programming, ESOP'10*, volume 6012 of *LNCS*, 2010.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 35(8), 1986.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd Int. Symp. on Programming*. Dunod, Paris, 1976.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2-3), 1992.

[CGG$^+$05] A. Costan, S. Gaubert, E. Goubault, M. Martel, and S. Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *Computer Aided Verification, CAV'05*, volume 3576 of *LNCS*, 2005.

[CR08] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Principles of Programming Languages, POPL'08*. ACM, 2008.

[GGTZ07] S. Gaubert, E. Goubault, A. Taly, and S. Zennou. Static analysis by policy iteration on relational domains. In *European Symposium on Programming, ESOP'07*, volume 4421 of *LNCS*, 2007.

[GS07a] T. Gawlitza and H. Seidl. Precise fixpoint computation through strategy iteration. In *European Symposium on Programming, ESOP'07*, volume 4421 of *LNCS*, 2007.

[GS07b] T. Gawlitza and H. Seidl. Precise relational invariants through strategy iteration. In *Computer Science Logic, CSL'07*, volume 4646 of *LNCS*, 2007.

[GS10] T. Gawlitza and H. Seidl. Computing relaxed abstract semantics w.r.t. quadratic zones precisely. In *Static Analysis Symposium, SAS'10*, volume 6337 of *LNCS*, 2010.

[HK66] A. J. Hoffman and R. M. Karp. On nonterminating stochastic games. *Management Sci.*, 12:359–370, 1966.

[How60] R. Howard. *Dynamic Programming and Markov Processes.* Wiley, 1960.

[HPR97] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2), August 1997.

[JAL] B. Jeannet, M. Argoud, and G. Lalire. The Interproc interprocedural analyzer. `http://pop-art.inrialpes.fr/interproc/interprocweb.cgi`.

[Jea] B. Jeannet. The BDDAPRON logico-numerical abstract domains library. `http://www.inrialpes.fr/pop-art/people/bjeannet/bjeannet-forge/bddapron%/`.

[Jea10] B. Jeannet. Some experience on the software engineering of abstract interpretation tools. In *Int. Workshop on Tools for Automatic Program AnalysiS, TAPAS'2010*, volume 267 of *ENTCS*. Elsevier, 2010.

[JM09] B. Jeannet and A. Miné. APRON: A library of numerical abstract domains for static analysis. In *Computer Aided Verification, CAV'2009*, volume 5643 of *LNCS*, 2009. `http://apron.cri.ensmp.fr/library/`.

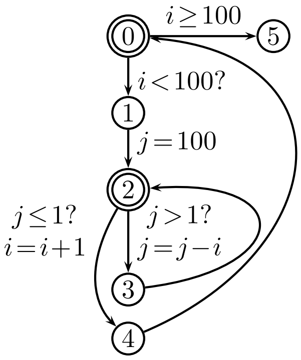
[Min02] A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *Verification, Model-Checking and Abstract Interpretation, VMCAI'06*, volume 3855 of *LNCS*, 2002.

[Min06] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1), 2006.

[Sha53] L. S. Shapley. Stochastic games. In *Proceedings of the National Academy of Sciences*, volume 39, pages 1095–1100, 1953.

[SSM05] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Verification, Model Checking, and Abstract Interpretation, VMCAI'05*, volume 3385 of *LNCS*, 2005.

A Comparison between box iteration and box policy iteration



Kleene box iteration			Local Kleene iteration for solving equations on π_0 = left policy for min	
	⓪ $i \in [1,1]$	① $i \in [1,1]$	⓪ $i \in [1,1]$	
	② $i \in [1,1]$	③ $i \in [1,1]$	① $i \in [1,99]$	③ $i \in [1,99]$
	$j \in [100,100]$	$j \in [99,99]$	$j \in [100,100]$	$j \in [1,99]$
	② $i \in [1,1]$	③ $i \in [1,1]$		
	$j \in [99,100]$	$j \in [98,99]$		
widen	② $i \in [1,1]$	③ $i \in [1,1]$		
	$j \in [-\infty,100]$	$j \in [1,99]$		
	④ $i \in [2,2]$			
	$j \in [-\infty,1]$			
	⓪ $i \in [1,2]$	① $i \in [1,2]$		
widen	⓪ $i \in [1,+\infty]$	① $i \in [1,99]$		
	② $i \in [1,99]$	③ $i \in [1,99]$	② $i \in [1,99]$	③ stable
	$j \in [-\infty,100]$	$j \in [1,99]$	$j \in [1,100]$	
widen	② $i \in [1,+\infty]$	③ $i \in [1,+\infty]$	④ $i \in [2,100]$	① stable
	$j \in [-\infty,100]$	$j \in [1,99]$		
	④ $i \in [2,+\infty]$			
	$j \in [-\infty,1]$			
(stable)				
narrow	⓪ $i \in [1,+\infty]$	① $i \in [1,99]$		
narrow	② $i \in [1,+\infty]$	③ $i \in [1,+\infty]$		
	$j \in [-\infty,100]$	$j \in [1,99]$		
(stable)	④ $i \in [2,+\infty]$	⑤ $i \in [100,+\infty]$	⑤ $i \in [100,100]$	
	$j \in [-\infty,1]$			
			no improvement with another policy	