Computing Quadratic Invariants with Min- and Max-Policy Iterations: a Practical Comparison

Pierre Roux[1,2] and Pierre-Loïc Garoche[1]

[1] ONERA – The French Aerospace Lab, Toulouse, FRANCE
[2] ISAE, Toulouse, FRANCE

Abstract. Policy iterations have been known in static analysis since a small decade. Despite the impressive results they provide – achieving a precise fixpoint without the need of widening/narrowing mechanisms of abstract interpretation – their use is not yet widespread. Furthermore, there are basically two dual approaches: min-policies and max-policies, but they have not yet been practically compared.
Multiple issues could explain their relative low adoption in the research communities: implementation of the theory is not obvious; initialization is rarely addressed; integration with other abstraction or fixpoint engine not mentionned; etc. This paper tries to present a Policy Iteration Primer, summarizing the approaches from the practical side, focusing on their implementation and use.
We implemented both of them for a specific setting: the computation of quadratic templates, which appear useful to analyze controllers such as found in civil aircrafts or UAVs.

Keywords: abstract interpretation, policy iteration, linear systems with guards, quadratic invariants, ellipsoids, semidefinite programming

1 Introduction

Abstract interpretation is now commonly used as a framework to describe static analyses of programs. The collecting semantics, i.e., set of reachable states, has first to be characterized as a fixpoint computation; then abstract domains allow to perform in the abstract the fixpoint computation and obtain a sound over-approximation of the concrete fixpoint.

The most famous approach of this fixpoint over-approximation is based on a Kleene fixpoint computation using widening and narrowing mechanisms [5]. The iteration process starts from an over-approximation, in the abstract domain, of the initial states, then it performs a sequence of computations using the abstract transfer function of the program. These iterations can be understood as local computations: each statement of the program is considered one by one until the global fixpoint is reached. Widening operators are then used while computing the iterates to ensure convergence. Narrowing helps to recover precision lost by widening steps: it is used once a postfixpoint is obtained to regain precision.

Another approach was more recently introduced in the static analysis community: policy[3] iterations [4,8,9]. The idea is to exactly solve the fixpoint equation for a given abstract domain when specific conditions are satisfied using appropriate mathematical solvers. For example when both the abstract domain and the fixpoint equation use linear equations, then linear programming could be used to compute the exact solution without the need of widening and narrowing [8,9]. Similarly when the function and the abstract domain are at most quadratic, semi-definite programming (SDP) could be used [1,11,12]. In practice, the abstract domains should be rephrased as template domains, i.e., a finite a-priori-known set of functions that will be bounded thanks to the mathematical solvers.

This second approach is also very useful when abstract domains are not fitted with a lattice structure. For example ellipsoids, are not fitted with such: usually, there is no smallest (for inclusion order) ellipsoid containing two other given ellipsoids. But given a (fixed) set of quadratic templates, policy iterations could bound them. Policy iterations over quadratic templates is then a good approach to compute such invariants, that are not well suited for Kleene iterations.

We are interested in analyzing control command software, more specifically the ones found in UAVs or civil aircrafts. Most of them are based on well known principles of control theory: linear controllers. In general these controllers do not admit simple linear inductive invariants, but control theorists know for long [3,16] that such systems are stable if and only if they admit a quadratic invariant. Therefore we are interested in computing these invariants on such linear systems.

Few static analysis work rely on quadratic invariants to bound linear systems [1,2,6,7,11,18,19]. In particular, ellipsoids of dimension two are used in the famous Astrée tool [6,7].

About policy iterations, two different "schools" exist in the static analysis community. The "French school" [1,4,8,12] offers to iterate on min-policies, starting from an over-approximation of a fixpoint and decreasing the bounds until the fixpoint is reached. The "German school" [9,10,12] in contrary operates on max-policies, starting from bottom and increasing the bounds until a fixpoint is reached. While the first can be interrupted at any point leaving a sound over-approximation, the second approach requires to wait until the fixpoint is reached to provide its result.

Clearly those two approaches rely on comparable fundamentals, but no work actually compares them in practice. Furthermore their description is highly theoretical and not supported by actual implementation performing analyses on code. A few issues, that particularly matter when targeting a practical implementation, were also not actually addressed such as the initial state of the iterations, the use of unsound tools to perform numerical computations or the integration with other abstractions.

This paper tries to give a practical definition for both approaches and presents our experiments to compare them when inferring quadratic invariants for linear controllers. All the analyses have been implemented and all results are obtain without any other information than the code.

Section 2 details the state of the art, i.e., the definition of template domains, min- and max-policies. Section 3 provides some details on our implementation

[3] The word *strategy* is also used in the literature for *policy*, with equivalent meaning.

since most of the policy iteration papers about quadratic templates do not provide any implementation readily applicable to actual code and therefore do not deal with template synthesis or soundness of the floating point computations. Finally, Section 4 presents our experimental results while a last section concludes.

2 State of the Art

The basic idea of policy iteration is to decompose fixpoint computation problems to enable the use of numerical optimization tools to compute bounds that are hard to guess for the widening or to retrieve via narrowing.

2.1 Template Domains

Policy iteration is performed on so called template domains. Given a finite set $\{ t_1, \dots, t_n \}$ of expressions on program variables $\mathbb{V}$, the template domain $\mathcal{T}$ is defined as $\overline{\mathbb{R}}^n = (\mathbb{R} \cup \{ -\infty, +\infty \})^n$ and the meaning of an abstract value $(b_1, \dots, b_n) \in \mathcal{T}$ is the set of environments

$$\gamma_{\mathcal{T}}(b_1, \dots, b_n) = \{\rho \in (\mathbb{V} \to \mathbb{R}) \mid [\![t_1]\!](\rho) \le b_1, \dots, [\![t_n]\!](\rho) \le b_n\}$$

where $[\![t_i]\!](\rho)$ is the result of the evaluation of expression t_i in environment ρ. In other words, the abstract value $(b_1, \dots, b_n)$ represents all the environments satisfying all the constraints $t_i \le b_i$.

Indeed, many common abstract domains can be rephrased as template domains. For instance the intervals domain is obtained with templates $-x_i$ and x_i for all variables $x_i \in \mathbb{V}$ and the octagon domain [17] by adding all the $\pm x_i \pm x_j$. The shape of the templates to be considered for policy iteration depends on the optimization tools used. For instance, linear programming [8,9] allows any linear templates whereas quadratic templates can be handled thanks to semidefinite programming and an appropriate relaxation [1,11,12]. This paper focuses on the latter case.

$$\begin{tikzcd}
\boxed{1} \arrow[r, "\begin{array}{l} x_0 := 0 \\ x_1 := 0 \\ x_2 := 0 \end{array}"] & \boxed{2} \arrow[loop right, "\begin{array}{l} -1 \le in \le 1 \, , \, x_0 := 0.9379\, x_0 - 0.0381\, x_1 - 0.0414\, x_2 + 0.0237\, in \\ x_1 := -0.0404\, x_0 + 0.968\, x_1 - 0.0179\, x_2 + 0.0143\, in \\ x_2 := 0.0142\, x_0 - 0.0197\, x_1 + 0.9823\, x_2 + 0.0077\, in \end{array}"]
\end{tikzcd}$$

Fig. 1. Control flow graph for our running example.

Example 1. To bound the variables of the program whose control flow graph is depicted on Figure 1, we use the quadratic template[4]: $t_1 := 6.2547x_0^2 + 12.1868x_1^2 + 3.8775x_2^2 - 10.61x_0x_1 - 2.4306x_0x_2 + 2.4182x_1x_2$. Templates $t_2 := x_0^2$, $t_3 := x_1^2$ and $t_4 := x_2^2$ are added in order to get bounds on each variable. Using those templates, policy iterations compute the invariant[5] $(1.0029,\ 0.1795,\ 0.1136,\ 0.2757) \in \mathcal{T}$, meaning: $t_1 \le 1.0029 \wedge x_0^2 \le 0.1795 \wedge x_1^2 \le 0.1136 \wedge x_2^2 \le 0.2757$ or equivalently: $t_1 \le 1.0029 \wedge |x_0| \le 0.4236 \wedge |x_1| \le 0.3371 \wedge |x_2| \le 0.5251$. This is a cropped ellipsoid as displayed on Figure 2.

[4] How this template was chosen will be explained later in Section 3.2.

[5] All figures are rounded to the fourth digit.

2.2 System of Equations

While Kleene iterations iterate locally through each construct of the program, policy iterations require a global view on the analyzed program. For that purpose, the whole program is first translated into a system of equations which is later solved.

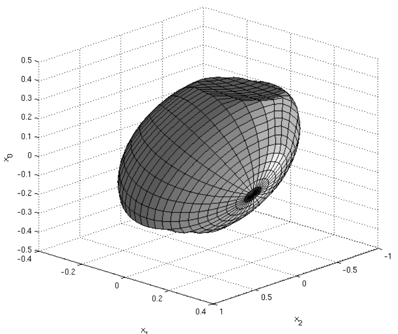

Fig. 2. Invariant for our running example.

Starting from the control flow graph of the analyzed program, a system of equations is defined with a variable $b_{i,j}$ for each vertex i of the graph and each template t_j.

Example 2. Here is the system of equations for our running example:

$$\begin{cases} b_{1,1} = +\infty \qquad b_{1,2} = +\infty \qquad b_{1,3} = +\infty \qquad b_{1,4} = +\infty \\ b_{2,1} = \max\{0 \mid \mathrm{be}(1)\} \vee \max\{r(t_1) \mid (-1 \leq in \leq 1) \wedge \mathrm{be}(2)\} \\ b_{2,2} = \max\{0 \mid \mathrm{be}(1)\} \vee \max\{r(t_2) \mid (-1 \leq in \leq 1) \wedge \mathrm{be}(2)\} \\ b_{2,3} = \max\{0 \mid \mathrm{be}(1)\} \vee \max\{r(t_3) \mid (-1 \leq in \leq 1) \wedge \mathrm{be}(2)\} \\ b_{2,4} = \max\{0 \mid \mathrm{be}(1)\} \vee \max\{r(t_4) \mid (-1 \leq in \leq 1) \wedge \mathrm{be}(2)\} \end{cases} \tag{1}$$

where be(i) denotes $(t_1 \leq b_{i,1}) \wedge (t_2 \leq b_{i,2}) \wedge (t_3 \leq b_{i,3}) \wedge (t_4 \leq b_{i,4})$ and $r(t)$ is the template t in which variable x_0 is replaced by $0.9379\, x_0 - 0.0381\, x_1 - 0.0414\, x_2 + 0.0237\, in$, variable x_1 is replaced by $-0.0404\, x_0 + 0.968\, x_1 - 0.0179\, x_2 + 0.0143\, in$ and variable x_2 is replaced by $0.0142\, x_0 - 0.0197\, x_1 + 0.9823\, x_2 + 0.0077\, in$. The usual maximum on $\overline{\mathbb{R}}$ is denoted $\vee$.

Each $b_{i,j}$ bounds the template t_j at program point i and is defined in one equation as a maximum over as many terms as incoming edges in i. More precisely, each edge between two vertices v and v' translates to a term in each equation $b_{v',j}$ on the pattern: $\max\left\{r(t_j) \,\middle|\, c \wedge \bigwedge_j (t_j \leq b_{v,j})\right\}$ where c and r are respectively the constraints and the assignments associated to this edge. This expresses the maximum value the template t_j can reach in destination vertex v' when applying the assignments r on values satisfying both the constraints c of the edge and the constraints $t_j \leq b_{v,j}$ of the initial vertex v. Finally, the program starting point is initialized to $(+\infty, \ldots, +\infty)$, meaning all equations for $b_{i_0,j}$, where i_0 is the starting point, become $b_{i_0,j} = +\infty$. Thus, for any solution $(b_{1,1}, \ldots, b_{1,n}, \ldots)$ of the equations, $\gamma_{\mathcal{T}}(b_{i,1}, \ldots, b_{i,n})$ is an overapproximation of reachable states of the program at point i.

2.3 Policy Iterations

Two different techniques can be found in the literature to compute an overapproximation of the least solution of the previous system of equations (which existence is proved thanks to Knaster-Tarski theorem).

Min-Policy Iterations To some extent, Min-Policy iterations [1] can be seen as a very efficient *narrowing*, since they perform descending iterations from a postfixpoint towards some fixpoint, working in a way similar to the Newton-Raphson numerical method. Iterations are not guaranteed to reach a fixpoint but can be stopped at any time leaving an overapproximation thereof. Moreover, convergence is usually fast.

Writing a system of equations $b = F(b)$ with $b = (b_{i,j})_{i \in [\![1,n]\!], j \in [\![1,p]\!]}$ and $F : \overline{\mathbb{R}}^{np} \to \overline{\mathbb{R}}^{np}$ (n being the number of templates and p the number of vertices in the control flow graph), a min-policy is defined as follows: $\underline{F}$ is a min-policy for F if for every $b \in \overline{\mathbb{R}}^{np}$, $F(b) \leq \underline{F}(b)$ and there exist some $b_0 \in \overline{\mathbb{R}}^{np}$ such that $\underline{F}(b_0) = F(b_0)$.

Example 3. Considering the system of one equation $b_{1,1} = 0 \vee \sqrt{b_{1,1}}$ where $\sqrt{x}$ is defined as $-\infty$ for negative numbers x, $\underline{F}$ defined as $\underline{F}(b) := 0 \vee \left(\frac{b_{1,1}}{8} + 2\right)$ is a min-policy. Indeed, for all $b_{1,1} \in \overline{\mathbb{R}}$, $F(b) = 0 \vee \sqrt{b_{1,1}} \leq 0 \vee \frac{b_{1,1}}{8} + 2 = \underline{F}(b)$, and for $b_0 = 16$, $F(b_0) = \sqrt{16} = \frac{16}{8} + 2 = \underline{F}(b_0)$. This is illustrated on Figure 3 on which $\sigma_1 = \underline{F}$.

The following theorem can then be used to compute the least fixpoint of F.

Theorem 1. *Given a (potentially infinite) set $\underline{\mathcal{F}}$ of min-policies for F. If for all $b \in \overline{\mathbb{R}}^{np}$ there exist a policy $\underline{F} \in \underline{\mathcal{F}}$ interpolating F at point b (i.e. $\underline{F}(b) = F(b)$) and if each $\underline{F} \in \underline{\mathcal{F}}$ has a least fixpoint $\mathrm{lfp}\underline{F}$, then the least fixpoint of F satisfies*

$$\mathrm{lfp}F = \bigwedge_{\underline{F} \in \underline{\mathcal{F}}} \mathrm{lfp}\underline{F}.$$

Iterations are done with two main objects: a min-policy σ and a tuple b of values for variables $b_{i,j}$ of the system of equations. The following policy iteration algorithm starts from some postfixpoint b_0 of F and aims at refining it to produce a better overapproximation of a fixpoint of F. Policy iteration algorithms always proceed by iterating two phases: first a policy σ_i is selected, then it is solved giving some b_i. More precisely in our case:

- find a linear min-policy σ_{i+1} being tangent to F at point b_i, this can be done thanks to a semi definite programming solver and a lagrangian relaxation;
- compute the least fixpoint b_{i+1} of policy σ_{i+1} thanks to linear programming.

Iterations can be stopped at any point (for instance after a fixed number of iterations or when progress between b_i and b_{i+1} is considered small enough) leaving an overapproximation b of a fixpoint of F.

Example 4. We perform min-policy iterations on the system of equation of Example 3.

- We start from the postfixpoint $b_0 = 16$. This postfixpoint could be obtained through Kleene iterations for instance.

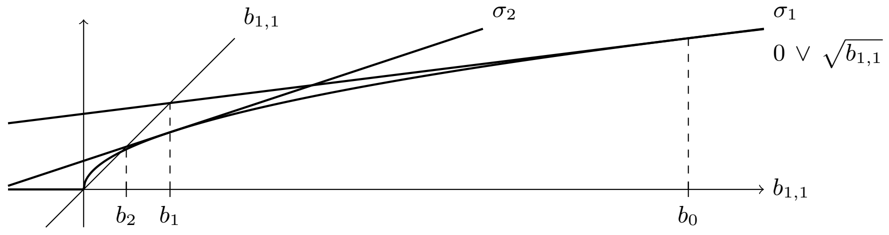

Fig. 3. Illustration of Example 4.

– For each term of the unique equation, we look for an hyperplane tangent to the term at point b_0. 0 is tangent to 0 at point b_0 and $\frac{b_{1,1}}{8} + 2$ is tangent to $\sqrt{b_{1,1}}$ at point b_0 (c.f., Figure 3), this gives the following linear min-policy: $\sigma_1 =$
$$\left\{ b_{1,1} = 0 \vee \left(\frac{b_{1,1}}{8} + 2\right)\right.$$
– The least fixpoint of σ_1 is then: $b_1 = \frac{16}{7} \simeq 2.2857$.
– Looking for hyperplanes tangent at point b_1 gives the min-policy: $\sigma_2 =$
$$\left\{ b_{1,1} = 0 \vee \left(\frac{\sqrt{7}}{8} b_{1,1} + \frac{2}{\sqrt{7}}\right)\right.$$
– Hence $b_2 = \frac{16}{8\sqrt{7}-7} \simeq 1.1295$.

These two first iterations are illustrated on Figure 3. The procedure then rapidly converges to the fixpoint $b_{1,1} = 1$ (the next iterates being $b_3 \simeq 1.0035$ and $b_4 \simeq 1.0000$) and can be stopped as soon as the accuracy is deemed satisfying.

Example 5. We perform min-policy iteration on the running example.

– We start from the postfixpoint $\beta_0 = (+\infty, +\infty, +\infty, +\infty, 1000000, +\infty, +\infty, +\infty)$, which could be obtained through Kleene iterations for instance.
– For each term of each equation, we look for an hyperplane tangent to the term at point b_0. This can be done thanks to a semi definite programming solver and gives the following linear min-policy:
$$\sigma_1 = \left\{\begin{array}{llll} b_{1,1} = +\infty & b_{1,2} = +\infty & b_{1,3} = +\infty & b_{1,4} = +\infty \\ b_{2,1} = 0 \vee 0.9857\, b_{2,1} + 0.0152 & & b_{2,2} = 0 \vee 0.2195\, b_{2,1} + 11.0979 & \\ b_{2,3} = 0 \vee 0.1143\, b_{2,1} + 4.8347 & & b_{2,4} = 0 \vee 0.2669\, b_{2,1} + 3.9796 & \end{array}\right.$$
– A linear programming solver allows to compute the least fixpoint of σ_1: $b_1 = (+\infty, +\infty, +\infty, +\infty, 1.0664, 11.3324, 4.9568, 4.2644)$.
– $$\sigma_2 = \left\{\begin{array}{llll} b_{1,1} = +\infty & b_{1,2} = +\infty & b_{1,3} = +\infty & b_{1,4} = +\infty \\ b_{2,1} = 0 \vee 0.9857\, b_{2,1} + 0.0143 & & b_{2,2} = 0 \vee 0.2302\, b_{2,1} + 0.0120 & \\ b_{2,3} = 0 \vee 0.1190\, b_{2,1} + 0.0052 & & b_{2,4} = 0 \vee 0.2708\, b_{2,1} + 0.0042 & \end{array}\right.$$
– $b_2 = (+\infty, +\infty, +\infty, +\infty, 1.0029, 0.2429, 0.1245, 0.2757)$.
– $$\sigma_3 = \left\{\begin{array}{llll} b_{1,1} = +\infty & b_{1,2} = +\infty & b_{1,3} = +\infty & b_{1,4} = +\infty \\ \multicolumn{4}{l}{b_{2,1} = 0 \vee 0.9857\, b_{2,1} + 0.0143} \\ \multicolumn{4}{l}{b_{2,2} = 0 \vee 0.0390\, b_{2,1} + 0.7426\, b_{2,2} + 0.0114} \\ \multicolumn{4}{l}{b_{2,3} = 0 \vee 0.0340\, b_{2,1} + 0.6635\, b_{2,3} + 0.0050} \\ \multicolumn{4}{l}{b_{2,4} = 0 \vee 0.2709\, b_{2,1} + 0.0040} \end{array}\right.$$

– $b_3 = (+\infty, +\infty, +\infty, +\infty, 1.0029, 0.1962, 0.1160, 0.2757)$.
– $\sigma_4 =$
$$\begin{cases} b_{1,1} = +\infty, \quad b_{1,2} = +\infty, \quad b_{1,3} = +\infty, \quad b_{1,4} = +\infty \\ b_{2,1} = 0 \vee 0.9857\, b_{2,1} + 0.0143 \\ b_{2,2} = 0 \vee 0.0194\, b_{2,1} + 0.8340\, b_{2,2} + 0.0104 \\ b_{2,3} = 0 \vee 0.0214\, b_{2,1} + 0.7688\, b_{2,3} + 0.0049 \\ b_{2,4} = 0 \vee 0.2709\, b_{2,1} + 0.0040 \end{cases}$$
– $b_4 = (+\infty, +\infty, +\infty, +\infty, 1.0029, 0.1803, 0.1137, 0.2757)$.

Two more iterations lead to $b_6 = (+\infty, +\infty, +\infty, +\infty,$ 1.0029, 0.1795, 0.1136, 0.2757) which is the invariant given in Example 1 and depicted on Figure 2.

Max-Policy Iterations Behaving somewhat as a super *widening*, Max-Policy iterations [11] work in the opposite direction compared to Min-Policy iterations. They start from bottom and iterate computations of greatest fixpoints on a set of max-policies until a global fixpoint is reached. Unlike the previous approach, this terminates with a *theoretically* precise fixpoint, but the user has to wait until the end since intermediate results are not overapproximations of a fixpoint.

Max-policies are the dual of min-policies: $\overline{F}$ is a max-policy for F if for every $b \in \overline{\mathbb{R}}^{np}$, $\overline{F}(b) \leq F(b)$ and there exist some $b_0 \in \overline{\mathbb{R}}^{np}$ such that $\overline{F}(b_0) = F(b_0)$. In particular, the choice of one term in each equation is a max-policy. From now on, only this last kind of max-policies will be considered.

Example 6. A max-policy of the system of equations from Example 2:

$$\begin{cases} b_{1,1} = +\infty,\ b_{1,2} = +\infty,\ b_{1,3} = +\infty,\ b_{1,4} = +\infty \\ b_{2,1} = \max\{r(t_1) \mid (-1 \leq in \leq 1) \wedge \mathrm{be}(2)\} \\ b_{2,2} = \max\{0 \mid \mathrm{be}(1)\} \\ b_{2,3} = \max\{0 \mid \mathrm{be}(1)\} \\ b_{2,4} = \max\{r(t_4) \mid (-1 \leq in \leq 1) \wedge \mathrm{be}(2)\} \end{cases}$$

Iterations are again done with two main objects: a max-policy σ and a tuple b of values for variables $b_{i,j}$ of the system of equations. Considering that computing a fixpoint on a given policy reduces to a mathematical optimization problem and that a fixpoint of the whole equation system is also a fixpoint of some policy, the following policy iteration algorithm aims at finding such a policy by solving optimization problems. To initiate the algorithm, a term $-\infty$ is added to each equation, the initial policy σ_0 is then $-\infty$ for each equation and the initial value b_0 is the tuple $(-\infty, \ldots, -\infty)$. Then policies are iterated:

– find a policy σ_{i+1} improving policy σ_i at point b_i, i.e. that reaches (strictly) greater values evaluated at point b_i; if none is found, exit;
– compute the greatest fixpoint b_{i+1} of policy σ_{i+1}.

The last tuple b is then a fixpoint of the whole system of equations.

Remark 1. Although min and max policies are dual concepts, we are in both cases looking for *over*approximations of the least fixpoint of the system of equations, thus the algorithms are *not* dual.

Example 7. We perform max-policy iterations on the running example. For that, we first add $-\infty$ terms to each equation, leading to the following system of equations:

$$\begin{cases} b_{1,1} = -\infty \vee +\infty \quad b_{1,2} = -\infty \vee +\infty \quad b_{1,3} = -\infty \vee +\infty \quad b_{1,4} = -\infty \vee +\infty \\ b_{2,1} = -\infty \vee \max\{0 \mid \mathrm{be}(1)\} \vee \max\{r(t_1) \mid (-1 \leq in \leq 1) \wedge \mathrm{be}(2)\} \\ b_{2,2} = -\infty \vee \max\{0 \mid \mathrm{be}(1)\} \vee \max\{r(t_2) \mid (-1 \leq in \leq 1) \wedge \mathrm{be}(2)\} \\ b_{2,3} = -\infty \vee \max\{0 \mid \mathrm{be}(1)\} \vee \max\{r(t_3) \mid (-1 \leq in \leq 1) \wedge \mathrm{be}(2)\} \\ b_{2,4} = -\infty \vee \max\{0 \mid \mathrm{be}(1)\} \vee \max\{r(t_4) \mid (-1 \leq in \leq 1) \wedge \mathrm{be}(2)\}. \end{cases}$$

– We start with initial policy $\sigma_0 =$

$$\begin{cases} b_{1,1} = -\infty \quad b_{1,2} = -\infty \quad b_{1,3} = -\infty \quad b_{1,4} = -\infty \\ b_{2,1} = -\infty \quad b_{2,2} = -\infty \quad b_{2,3} = -\infty \quad b_{2,4} = -\infty. \end{cases}$$

– Its greatest fixpoint is $b_0 = (-\infty, -\infty, -\infty, -\infty, -\infty, -\infty, -\infty, -\infty)$.
– We now look for a policy σ_1 improving σ_0 at point b_0. For the first four equations, the term $+\infty$ is definitely greater than $-\infty$. The strategy $\sigma_1 =$

$$\begin{cases} b_{1,1} = +\infty \quad b_{1,2} = +\infty \quad b_{1,3} = +\infty \quad b_{1,4} = +\infty \\ b_{2,1} = -\infty \quad b_{2,2} = -\infty \quad b_{2,3} = -\infty \quad b_{2,4} = -\infty. \end{cases}$$

is then a suitable choice.
– Hence $b_1 = (+\infty, +\infty, +\infty, +\infty, -\infty, -\infty, -\infty, -\infty)$.
– We again look for a policy σ_2 improving σ_1 at point b_0. There is nothing strictly greater than $+\infty$ in $\overline{\mathbb{R}}$ and we keep the $+\infty$ terms for the first four equations. In the four remaining equations, replacing the $b_{i,j}$ with values from b_1 in be(1) and be(2) respectively gives formula equivalent to *true* and *false*. This way, for these four equations, the first term reduces to 0 whereas the second term evaluates to $-\infty$. 0 being greater than the $-\infty$ from b_1, we get an improving strategy $\sigma_2 =$

$$\begin{cases} b_{1,1} = +\infty \quad b_{1,2} = +\infty \quad b_{1,3} = +\infty \quad b_{1,4} = +\infty \\ b_{2,1} = \max\{0 \mid \mathrm{be}(1)\} \quad b_{2,2} = \max\{0 \mid \mathrm{be}(1)\} \\ b_{2,3} = \max\{0 \mid \mathrm{be}(1)\} \quad b_{2,4} = \max\{0 \mid \mathrm{be}(1)\}. \end{cases}$$

– $b_2 = (+\infty, +\infty, +\infty, +\infty, 0, 0, 0, 0)$.
– Now that the $b_{2,j}$ in b_2 are no longer $-\infty$, be(2) is no longer *false* and it becomes interesting to select the second terms in the four last equations, hence $\sigma_3 =$

$$\begin{cases} b_{1,1} = +\infty \quad b_{1,2} = +\infty \quad b_{1,3} = +\infty \quad b_{1,4} = +\infty \\ b_{2,1} = \max\{r(t_1) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\} \\ b_{2,2} = \max\{r(t_2) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\} \\ b_{2,3} = \max\{r(t_3) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\} \\ b_{2,4} = \max\{r(t_4) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\}. \end{cases}$$

– The greatest fixpoint $b_3 = (+\infty, +\infty, +\infty, +\infty,\ 1.0077,\ 0.1801,\ 0.1141,\ 0.2771)$ of σ_3 can be computed thanks to a semi-definite programming solver and an appropriate relaxation.
– No more improving strategy.

After four iterations, the algorithm has found the same least fixpoint than min policies in Example 5.

The Max-Policy iteration builds an ascending chain of abstract elements similarly to Kleene iterations elements. However it is guaranteed to be finite, bounded by the number of policies σ, while Kleene iterations require the use of widening to ensure termination. Since there are exponentially many max-policies in the number of templates and points of the control flow graph and since each policy can be an improving one only once, we have an exponential bound on the number of iterations. But in practice, only a small number of policies are usually considered and the number of iterations remains reasonable.

3 Implementation Details

This Section highlights a few features of our implementation of min- and max-policy iterations to compute quadratic invariants on linear systems. Some are just simple hacks to improve analysis performances. Others were needed to achieve full automaticity, ensure the soundness of the result or just to get any result at all on our benchmarks.

3.1 Control Flow Graph

In this paper, we only dealt with control flow graphs from which system of equations are extracted for policy iterations. In a traditional, abstract interpretation based, static analyzer, abstractions are computed by *abstract domains* [14] not having access to the whole control flow graph of the analyzed program but only to individual operations it performs. A symbolic abstract domain was then designed to rebuild the control flow graph. This way, policy iterations are packed in an abstract domain which can be used in a static analyzer through the same interface than any other numerical relational domain such as polyhedra or octagons for instance [15]. Full technical details on this point are unfortunately outside the scope of this paper. We refer the interested reader to [20] for more details.

3.2 Templates

Template domains used by policy iteration require templates to be given prior to the analyses. This greatly limits the automaticity of the method. However, heuristics can be designed for linear systems of the form $x_{k+1} = Ax_k + Bu_k$, like our running example. Those are ubiquitous in control applications where the vector x represents the internal state of the controller and u a bounded input.

This section first focuses on generating templates for pure linear systems then for guarded linear systems given as a control flow graph.

Pure Linear Systems Control theorists know for long [3,16] that such a system is stable (i.e. that x is bounded) if and only if the Lyapunov equation

$$P - A^T P A \succeq 0 \tag{2}$$

admits a symmetric positive definite matrix P as solution, where $M \succeq 0$ means that the matrix M is positive definite (i.e. for all x, $x^T M x \geq 0$). The template $t := x^T P x$ is then a quadratic template and policy iteration can be used to compute a bound b such that $t \leq b$ is an invariant of the system. This invariant is an ellipsoid [22].

Inequality (2) is a so called Linear Matrix Inequality (LMI) which can be solved thanks to a semidefinite programming solver. However, taking any random solution may lead to very grossly overapproximated invariants. It would be interesting to constrain more the set of solutions, for instance by forcing them to lie in a sphere as small as possible. More precisely, we will look for an ellipsoid P included in the smallest possible sphere and which is stable, i.e., such that

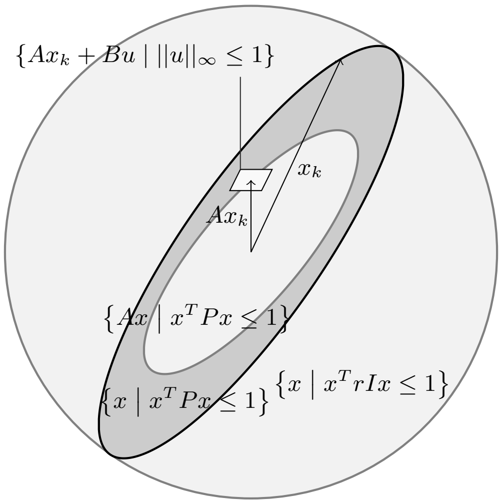
$\{Ax_k + Bu \mid ||u||_\infty \leq 1\}$
x_k
Ax_k
$\{Ax \mid x^T P x \leq 1\}$
$\{x \mid x^T P x \leq 1\}$
$\{x \mid x^T r I x \leq 1\}$

Fig. 4. Looking for an invariant ellipsoid included in the smallest possible sphere by maximizing r.

$$\forall x, \forall u, \left(||u||_\infty \leq 1 \wedge x^T P\, x \leq 1\right) \Rightarrow (Ax + Bu)^T\, P\, (Ax + Bu) \leq 1.$$

This is illustrated in Figure 4. The previous condition can be rewritten

$$\forall x, \forall u, \left(\left(\bigwedge_{i=0}^{p-1} \left(e_i^T u\right)^2 \leq 1\right) \wedge x^T P\, x \leq 1\right) \Rightarrow (Ax + Bu)^T\, P\, (Ax + Bu) \leq 1.$$

where e_i is the i-th vector of the canonical basis (i.e., with all coefficients equal to 0 except the i-th one which is 1). This amounts to: $\forall x, \forall u$,

$$\left(\bigwedge_{i=0}^{p-1} \begin{bmatrix} x \\ u \end{bmatrix}^T \begin{bmatrix} 0 & 0 \\ 0 & E^{i,i} \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \leq 1\right) \wedge \begin{bmatrix} x \\ u \end{bmatrix}^T \begin{bmatrix} P & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \leq 1 \Rightarrow \begin{bmatrix} x \\ u \end{bmatrix}^T \begin{bmatrix} A^T P A & A^T P B \\ B^T P A & B^T P B \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \leq 1$$

where $E^{i,j}$ is the matrix with 0 everywhere except the coefficient at line i, column j which is 1. Using a lagrangian relaxation, this holds when there are τ and $\lambda_0, \ldots, \lambda_{p-1}$ all positives such that

$$\begin{bmatrix} -A^T P A & -A^T P B & 0 \\ -B^T P A & -B^T P B & 0 \\ 0 & 0 & 1 \end{bmatrix} - \tau \begin{bmatrix} -P & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \textstyle\sum_{i=0}^{p-1} \lambda_i \begin{bmatrix} 0 & 0 & 0 \\ 0 & -E^{i,i} & 0 \\ 0 & 0 & 1 \end{bmatrix} \succeq 0 \qquad (3)$$

This is not an LMI since τ and P are both variables which means it cannot be directly solved 'as is'. However, there is a $\tau_{min} \in (0, 1)$ such that this inequality admits as solution a positive definite matrix P if and only if $\tau \in (\tau_{min}, 1)$. This value τ_{min} can then be efficiently approximated thanks to a dichotomy. It now remains to choose the 'best' τ in this interval. For this purpose, P is forced to be contained in the smallest possible sphere by maximizing r in the additional constraint

$$P \succeq rI. \tag{4}$$

We denote f the function mapping $\tau \in (\tau_{min}, 1)$ to the optimal value of the following semi definite program:

$$\begin{array}{ll} \text{maximize} & r \\ \text{subject to} & (3), (4), P^T = P, \bigwedge_{i=0}^{p-1} (\lambda_i > 0) \end{array}$$

Thus, this function can be evaluated for a given input τ simply by solving the above semi definite program. f is then sampled for some equally spaced values in the interval $(\tau_{min}, 1)$ and the matrix P obtained for the value enabling the maximum r is kept.

Example 8. With the following matrices A and B of the running example:

$$A := \begin{bmatrix} 0.9379 & -0.0381 & -0.0414 \\ -0.0404 & 0.968 & -0.0179 \\ 0.0142 & -0.0197 & 0.9823 \end{bmatrix} \qquad B := \begin{bmatrix} 0.0237 \\ 0.0143 \\ 0.0077 \end{bmatrix},$$

five steps of dichotomy give $\tau_{min} = 0.9921875$. Then computing the function f for a dozen of values between τ_{min} and 1, the following matrix P is selected, corresponding to $\tau = 0.9921875$:

$$P = \begin{bmatrix} 6.2547 & -5.3050 & -1.2153 \\ -5.3050 & 12.1868 & 1.2091 \\ -1.2153 & 1.2091 & 3.8775 \end{bmatrix}.$$

This is the template used in Example 1.

Guarded Linear Systems From a control flow graph, matrices A and B are extracted by looking at the strongly connected component of the relation "variable x linearly depends on variable y". Templates are then generated as above for these matrices. This is a pure heuristic since existence of templates for such subsystems does not mean that they will allow to bound the whole system, not even that it is stable. However, this is a reasonable choice since actual systems are usually designed around a pure linear core.

Finally, as seen in the running example, we add templates x^2 for each variable modified by the program. In the literature [1,11,12], templates x and $-x$ are used. Since results are usually symmetrical in our context (i.e. the same bound b is obtained for both templates: $x \leq b$ and $-x \leq b$), templates x^2 yield the same result (i.e. $x^2 \leq b^2$) making use of two times less templates for policy iteration, hence saving on computation costs.

3.3 Initial Value

In the policy iteration literature, system of equations require extra terms with initial values for each template at loop head. Although

those values do not come totally out of the blue, computing them does not appear absolutely obvious. As seen in the running example, we chose to replace them by an initial vertex (vertex 1 in Figure 1) initialized with bound $+\infty$ for each template and linked to loop head (vertex 2 in Figure 1) by an edge with initialization code. Thus, previous initial values for each template will actually be computed by policy iteration.

Considering policy iteration themselves, max-policies start from $(-\infty, \ldots, -\infty)$ whereas min-policies need to start from a postfixpoint. Such a postfixpoint could be computed through Kleene iterations using a simple widening with thresholds. However, just starting from a big value (for instance 10^6) for the quadratic templates computed in the previous Section and $+\infty$ for all others often yields in practice the same results at a lower cost.

3.4 Interval Constraints

To enable the use of semidefinite programming solvers, a relaxation must be used. It basically amounts to the following theorem.

Theorem 2 (Lagrangian relaxation). *Assume f and $g_1, \ldots, g_k$ functions $\mathbb{R} \to \mathbb{R}$, if there exist $\lambda_1, \ldots, \lambda_k \in \mathbb{R}$ all non negative such that.*

$$\forall x, f(x) - \sum_i \lambda_i g_i(x) \geq 0 \tag{5}$$

then

$$\forall x, \left(\bigwedge_i g_i(x) \geq 0 \right) \Rightarrow f(x) \geq 0. \tag{6}$$

Semidefinte programming solvers being unable to directly handle Equation (6), they are feeded with Equation (5). This usually works well, however the converse of Theorem 2 does not generally holds. In particular with a quadratic objective f and two linear constraints g_1 and g_2.

Example 9. We want to apply a relaxation on $x \in [1, 3] \Rightarrow -x^2 + 4x + 5$, that is Equation (6) with $f := x \mapsto -x^2 + 4x + 5$, $g_1 := x \mapsto x - 1$ and $g_2 := 3 - x$. Equation (5) then boils down to: $\forall x, -x^2 + (4 - \lambda_1 - \lambda_2)x + (5 + \lambda_1 - 3\lambda_2) \geq 0$. Unfortunately, not any $\lambda_1, \lambda_2 \in \mathbb{R}$ satisfy this. This is depicted on left of Figure 5.

This case is commonly encountered in practice, for instance with initial values of a program living in some range or with inputs bounded by an interval. Replacing the two linear constraints by an equivalent quadratic one constitutes an efficient workaround.

Example 10. When constraints $x - 1 \geq 0$ and $3 - x \geq 0$ are replaced by the equivalent $1 - (x - 2)^2 \geq 0$, relaxation works just fine (with relaxation coefficient $\lambda = 1$ for instance). This is depicted on right of Figure 5.

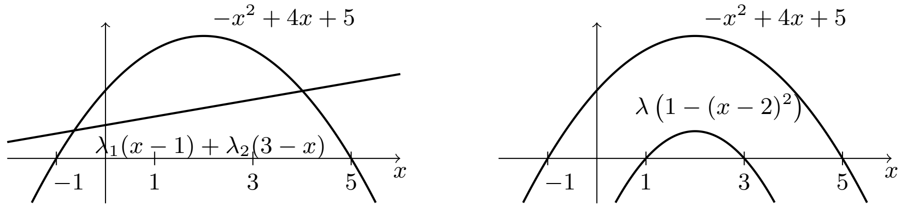

Fig. 5. Relaxation of interval constraints.

3.5 Soundness of the Result

For the sake of efficiency, the semidefinite programming solvers we use perform all their computations on floating point numbers and do not offer any strict soundness guarantee on their results.

To address this issue, we adopt the following strategy:

- first perform policy iterations with unsound solvers, just padding the equations to hopefully get a correct result;
- then check the soundness of previous result.

Padding the equations means for min-policies multiplying each temporary result β_i by $(1+\epsilon)$ for some small ϵ. For max-policies, all equations $\max\{p \mid q \leq c\}$ are basically replaced by $\max\{(1+\epsilon)p \mid q \leq (1+\epsilon)c\}$. In practice, while using solvers trying to achieve an accuracy of 10^{-8} on their results, a value of 10^{-4} for ϵ appears to be a good choice. The induced loss of accuracy on the final result is considered acceptable since bounds finally computed by our analysis are usually found to be at least a few percent larger than the actual maximal values reachable by the program. Finding a good way to padd equations to get correct results, while still preserving the best accuracy, however remains some kind of black magic.

Checking that a result is an actual postfixpoint amounts, for each term of the equation system, and after some relaxation[6], to prove that a given matrix is actually positive definite. This is done by carefully bounding the rounding error on a floating point Cholesky decomposition [23]. Proof of positive definiteness of an $n \times n$ matrix can then be achieved with $\mathrm{O}\left(n^3\right)$ floating point operations, which in practice induces only a very small overhead to the whole analysis.

Finally, a quick and dirty hack to recover a correct result in the rare event where the aforementioned soundness check fails consists in multiplying the — probably false — result by a small constant (for instance 1.1) and checking again its soundness. This sometimes enable to get a better result than $\top$, despite the first check failure, at the very low cost of an additional check.

Although all this gives satisfying results. It would remain interesting to compare the cost/accuracy trade off when using the verified solver VSDP [13] as already offered in the literature [1].

[6] This relaxation being the same than the one used during policy iterations, it doesn't introduce further conservatism by itself.

4 Experimental Results

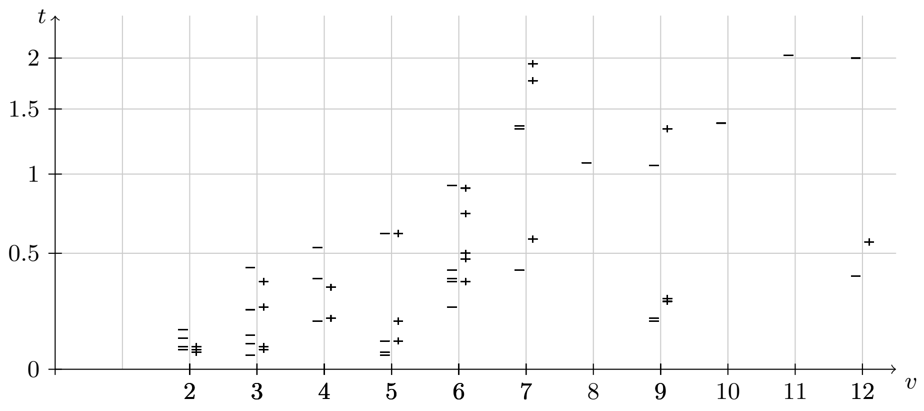

Fig. 6. Time (t in seconds) spent performing min ($-$ signs) and max ($+$ signs) policy iterations depending on the number v of variables in the analyzed program. Less $+$ than $-$ in a column indicate a failure of max-policies on a benchmark.

All the elements presented in this paper have been implemented as a new abstract domain in our static analyzer. Experiments were conducted on a set of stable linear systems. These systems were extracted from [1,7,22,24]. We have to recall to the reader that those systems, despite their apparent simplicity, do not admit simple linear invariants. Figure 6 compares analysis times with min and max-policy iterations. All computations were performed on an Intel Core2 @ 2.66GHz. The analyzer is released under a GPL license and available along with all examples and results at `http://cavale.enseeiht.fr/policy2014/`.

Figure 6 only gives times for policy iterations. Total analysis times also includes building the control flow graph and the equation system, computing appropriate templates and eventually checking soundness of the result. Time needed for control flow graph construction and soundness checking is very small compared to the time spent in policy iterations, whereas computing templates takes the same amount of magnitude in time than min-policies iteration.

For min-policies, the number of iterations performed lies between 3 and 7 when the stopping criterion is a relative progress below 10^{-4} between two consecutives β_i. For max-policies, the number of iterations was between 4 and 7.

Results obtained with min- and max-policies were the same. However, paradoxically enough, min-policies yield slightly more precise results. It is also worth noting that max-policies were in a few cases unable to produce a sound result whereas min-policies did. Finally, regarding the quality of the result, in cases where the maximum reachable values are known [24], bounds given by our analyzer seem to be accurate and are in average a few percents larger.

Finally, as seen on Figure 6, computation time for min and max-policies are comparable for small number of variables whereas min-policies scale way better for a larger number of variables. This can be explained by min-policies solving smaller semidefinite programming problems [12, Conclusion]. Therefore, we made min-policies the default in our tool.

5 Conclusion and Future Work

We have presented the two approaches to compute policy iterations: min- and max-policies, and we have instantiated them on quadratic templates using SDP solvers. Our implementation is then able to use both approaches and was applied on a series of representative examples of linear controllers.

This paper proposed a presentation of those two techniques from the tool implementation perspective. We also addressed mutiple issues that, for our point of view, prevent the development of these techniques: how to initialize the analysis? how to identify meaningful templates for a given problem? how to check the soundness of the computation when using tools relying on floating point implementation?

Our approch was implemented and actually integrated within a regular Kleene-based fixpoint abstract interpreter. It shows that the use of policy iteration in a more classic tool is accessible and could leverage the set of domains to perform analyses.

Amongst the results we obtain with our experimentations, one can notice that we obtain the same results with both approaches. Max-iteration were theoretically proved to provide the exact fixpoint but such proof was not stated for min-iteration. In practice – and in our setting – they give the same results.

However min-strategies showed to scale better, as expected. We have however to stress again that this may not be the case for other setting like the use of linear programming. Our experiments were only computed with quadratic templates on linear systems.

In terms of future work, different directions are open. First, the floating point semantics of analyzed programs has to be taken into account (instead of the real numbers semantics currently used). Second, it would be interesting to perform so called closed loop analyses of controllers, i.e., controllers considered with a model of their environment (so called *plant* for control theorists). Finally, since we have a prototype, it would be interesting to extend the kind of templates analyzable with policy iterations. Bernstein polynomials can be used to bound polynomial templates (beyond quadratic ones) [21]. Injecting this domain in the current setting could enable the analysis of a much wider class of programs. A deeper comparison of min- and max-policy should also consider an implementation with linear templates.

References

1. Assalé Adjé, Stéphane Gaubert, and Éric Goubault. Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. In *ESOP*, 2010.
2. Fernando Alegre, Éric Féron, and Santosh Pande. Using ellipsoidal domains to analyze control systems software. 2009. http://arxiv.org/abs/0909.1977.

3. Stephen Boyd, Laurent El Ghaoui, Éric Féron, and Venkataramanan Balakrishnan. *Linear Matrix Inequalities in System and Control Theory*, volume 15 of *SIAM*. Philadelphia, PA, June 1994.
4. Alexandru Costan, Stephane Gaubert, Eric Goubault, Matthieu Martel, and Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *CAV*, 2005.
5. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
6. Jérôme Feret. Static analysis of digital filters. In *ESOP*, number 2986, 2004.
7. Jérôme Feret. Numerical abstract domains for digital filters. In *International workshop on Numerical and Symbolic Abstract Domains (NSAD)*, 2005.
8. Stephane Gaubert, Eric Goubault, Ankur Taly, and Sarah Zennou. Static analysis by policy iteration on relational domains. In *ESOP*, 2007.
9. Thomas Gawlitza and Helmut Seidl. Precise fixpoint computation through strategy iteration. In *ESOP*, 2007.
10. Thomas Gawlitza and Helmut Seidl. Precise relational invariants through strategy iteration. In *CSL*, 2007.
11. Thomas Martin Gawlitza and Helmut Seidl. Computing relaxed abstract semantics w.r.t. quadratic zones precisely. In *SAS*, 2010.
12. Thomas Martin Gawlitza, Helmut Seidl, Assalé Adjé, Stéphane Gaubert, and Eric Goubault. Abstract interpretation meets convex optimization. *J. Symb. Comput.*, 47(12), 2012.
13. Christian Jansson, Denis Chaykin, and Christian Keil. Rigorous error bounds for the optimal value in semidefinite programming. *SIAM J. Numerical Analysis*, 46(1), 2007.
14. Bertrand Jeannet. Some experience on the software engineering of abstract interpretation tools. *Electr. Notes Theor. Comput. Sci.*, (2), 2010.
15. Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, 2009.
16. Aleksandr Mikhailovich Lyapunov. Problème général de la stabilité du mouvement. *Annals of Mathematics Studies*, 17, 1947.
17. Antoine Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, October 2001.
18. David Monniaux. Compositional analysis of floating-point linear numerical filters. In *CAV*, 2005.
19. Mardavij Roozbehani, Éric Féron, and Alexandre Megretski. Modeling, optimization and computation for software verification. In *HSCC*, 2005.
20. Pierre Roux and Pierre-Loïc Garoche. Integrating policy iterations in abstract interpreters. In *ATVA*, 2013.
21. Pierre Roux and Pierre-Loïc Garoche. A polynomial template abstract domain based on bernstein polynomials. In *NSV*, 2013.
22. Pierre Roux, Romain Jobredeaux, Pierre-Loïc Garoche, and Éric Féron. A generic ellipsoid abstract domain for linear time invariant systems. In *HSCC*. ACM, 2012.
23. Siegfried M. Rump. Verification of positive definiteness. *BIT Numerical Mathematics*, 46, 2006.
24. Yassamine Seladji and Olivier Bouissou. Numerical abstract domain using support functions. In *NFM*, 2013.