Integrating Policy Iterations in Abstract Interpreters *

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

[1] ONERA – The French Aerospace Lab, Toulouse, FRANCE
{pierre.roux,pierre-loic.garoche}@onera.fr
[2] ISAE, University of Toulouse, Toulouse, FRANCE

Abstract. Among precise abstract interpretation methods developed during the last decade, policy iterations is one of the most promising. Despite its efficiency, it has not yet seen a broad usage in static analyzers. We believe the main explanation to this restrictive use, beside the novelty of the technique, lies in its lack of integration in the classic abstract domain framework. This prevents an easy integration in existing static analyzers and collaboration with other, already implemented, abstract domains through reduced product. This paper aims at providing a classic abstract domain interface to policy iterations.
Usage of semidefinite programming to infer quadratic invariants on linear systems is one of the most appealing use of policy iteration. Combination with a template generation heuristic, inspired from existing methods from control theory, gives a fully automatic abstract domain to infer quadratic invariants on linear systems with guards. Those systems often constitute the core of embedded control systems and are hard, when not impossible, to analyze with linear abstract domains. The method has been implemented and applied to some benchmark systems, giving good results.

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

1 Introduction

Classic abstract interpretation based static analysis [8] heavily relies on the so called *widening*. This operator discards some information in order to enforce termination of the analysis. A *narrowing* can then partly recover this lost information. These heuristics often enable a good trade-off between cost and precision of analyses. However, even if impressive improvements were made in the last decade to widening [3,11,21,28, and references therein] and narrowing [22], they do not always guarantee precise results.

Another approach, that appeared in the last decade in the software verification community [7, for instance], is the use of dedicated mathematical solvers like linear or semidefinite programming as a way to solve some kind of problems in a verification setting. This led to the definition of so-called *policy iterations* [1,6,14,15,18,19], as another way to perform overapproximation but trying to achieve better precision than widening-based analyses.

* This work has been partially supported by the FNRAE Project CAVALE and the ANR INS Project CAFEIN.

However, even if promising, policy iterations have had very little impact on existing tools yet: their use seems orthogonal to the classic use of abstract domains in a Kleene setting, where reduced products allow domains to exchange knowledge about the system during computation.

An explanation to the lack of integration of policy iterations with Kleene-based analyses is that they need to work on a global view of the analyzed system, typically a control flow graph representation of it, while Kleene-based analyzers iterate through program points without providing a global view of the program to the abstract domains. Our solution is mainly to compute this graph while iterating through the program points with a Kleene-based analysis. Once the graph is obtained it can easily be used by policy iterations to compute numerical properties about the program. Those properties can then be exported to other domains through a reduced product. Moreover this new abstract domain can be applied on a strict subset of program variables abstracting other variables by information obtained through reduced product from other domains[3], thus allowing a true interplay between policy iterations and existing abstract domains.

Our proposal has been instantiated on the analysis of linear systems with guards admitting quadratic inductive invariants. These linear systems are widely present in critical embedded systems like aerospace control-command software but are hard to analyze with most abstract domains since they usually do not admit simple linear inductive invariants. The use of our framework enables a fully automatic analysis of such systems, relying on policy iterations with semidefinite programming, while other approaches either impose stronger restrictions on the class of analyzable programs or require more parameters to enable the analysis. It has been implemented and gave significant results.

After a brief policy iteration primer, the paper is organized as follows:

- Section 3 offers an *abstract domain rebuilding the control flow graph* through classic Kleene-based analysis;
- Section 4 enables the *embedding of policy iteration in an abstract domain* based on this computed graph and on template domains;
- in Section 5, we *automatically synthesize meaningful templates* for a specific class of programs: guarded linear systems admitting quadratic invariants.

The paper also provides, in Section 6, experimental results computed using our implementation of the analysis.

2 State of the Art – a Policy Iteration Primer

2.1 A Toy Imperative Language

Throughout this paper, a very classic toy imperative language will be used to illustrate our abstract domains. Figure 1 presents a program in this language.

Syntax A program of the language is a statement *stm* in the following grammar:

[3] As done with expensive relational domains in the abstract interpreter Astrée [9].

$stm ::= stm;\ stm \mid v := expr \mid v := ?(r,\ r) \mid \textbf{while}\ expr \leq r\ \textbf{do}\ stm\ \textbf{od}$
$\quad\quad \mid \textbf{if}\ expr \leq r\ \textbf{then}\ stm\ \textbf{else}\ stm\ \textbf{fi}$
$expr ::= v \mid r \mid expr + expr \mid expr - expr \mid expr \times expr$

with $v \in \mathbb{V}$, a set of variables, and $r \in \mathbb{R}$. $?(r_1,\ r_2)$ represents a random choice of a real number between r_1 and r_2 (useful to simulate inputs).

Collecting Semantics In the later, we denote by $[\![e]\!](\rho) \in \mathbb{R}$, the usual collecting semantics of an expression e in an environment $\rho : \mathbb{V} \to \mathbb{R}$; and by $[\![s]\!](R) \subseteq (\mathbb{V} \to \mathbb{R})$ the collecting semantics of a statement s for a set of environments $R \subseteq (\mathbb{V} \to \mathbb{R})$.

```
x0 := 0; x1 := 0; x2 := 0;
while −1 ≤ 0 do
  in := ?(−1, 1);
  x0' := x0; x1' := x1; x2' := x2;
  x0 := 0.9379 x0'−0.0381 x1'−0.0414 x2'+0.0237 in;
  x1 := −0.0404 x0'+0.968 x1'−0.0179 x2'+0.0143 in;
  x2 := 0.0142 x0'−0.0197 x1'+0.9823 x2'+0.0077 in;
od
```

Fig. 1. Example of program.

It is worth noting that this semantics is given with operations over real numbers $\mathbb{R}$ whereas an actual program would compute using floating point values. This issue will not be addressed in this paper and is left as future work.

2.2 Kleene Iterations with Widening and Narrowing

The previous concrete semantics being non computable, the basic idea of abstract interpretation is to compute a so called abstract semantics. This abstract semantics is designed as a computable overapproximation of the concrete one.

Abstract domains constitute the basic bricks of abstract interpreters. They are given by a complete lattice $\mathcal{D}$, a concretization function $\gamma_{\mathcal{D}} : \mathcal{D} \to 2^{\mathbb{V}\to\mathbb{R}}$ and computable abstract operators $[\![v := e]\!]^\sharp$, $[\![v := ?(\mathrm{r}_1, \mathrm{r}_2)]\!]^\sharp$ and $[\![e \leq r]\!]^\sharp : \mathcal{D} \to \mathcal{D}$. The concretization function gives a concrete meaning to each abstract value in $\mathcal{D}$ by mapping it to the set of environments it abstracts. The abstract semantics $[\![.]\!]^\sharp$ is then defined by replacing semantics of assignments and guards with the corresponding abstract operator in the equations of the previous section. This abstract semantics can thus be computed, provided fixpoints are reachable after finitely many iterations of loop bodies' semantics. Assuming some soundness hypotheses on the abstract operators, the abstract semantics of a program p can be proved to be an overapproximation of the concrete one, that is $[\![p]\!] \subseteq \gamma_{\mathcal{D}}([\![p]\!]^\sharp)$.

An operator called widening is used to ensure convergence in finitely many iterations by giving up some precision. Some of this lost precision can then be retrieved by descending iterations with a so called narrowing. However, this does not always regain all of it.

Example 1. Analyzing the program of Figure 1 with the intervals domain, we get after a first iteration of the loop $x_0 \in [-0.0237, 0.0237] \wedge x_1 \in [-0.0143, 0.0143] \wedge x_2 \in [-0.0077, 0.0077]$. After a second iteration $x_0 \in [-0.0467, 0.0467] \wedge x_1 \in [-0.0292, 0.0292] \wedge x_2 \in [-0.0158, 0.0158]$,... This does not converge, simply because the program does not admit any invariant in the intervals domain. Unlike the intervals domain, invariants exist in the polyhedra domain. However, classic Kleene iterations with this domain are in practice unable to compute any.

2.3 Policy Iterations

The basic idea of policy[4] iteration is to use numerical optimization tools to compute those bounds that are hard to guess for the widening or to retrieve via narrowing.

Another advantage of the method is to abstract sequences of program instructions like loop bodies "en bloc", avoiding intermediate abstractions which can cause irreversible losses of precision[5].

Template Domains Policy iteration is performed on so called template domains. Given a finite set $\{t_1, \ldots, t_n\}$ of expressions on 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 $(b_1, \ldots, b_n) \in \mathcal{T}$ is the set of environments $\gamma_{\mathcal{T}}(b_1, \ldots, b_n) = \{\rho \in (\mathbb{V} \to \mathbb{R}) \mid [\![t_1]\!](\rho) \leq b_1, \ldots, [\![t_n]\!](\rho) \leq b_n\}$. In other words, the abstract value $(b_1, \ldots, b_n)$ represents all the environments satisfying all the constraints $t_i \leq b_i$.

Indeed, many common abstract domains are 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 [24] 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 [14,16] allows any linear templates whereas quadratic templates can be handled thanks to semidefinite programming and an appropriate relaxation [1,18,19].

Example 2. To bound the variables of the program of Figure 1, we use the quadratic template[6]: $t_1 := 6.2547x_0^2 + 12.1868x_1^2 + 3.8775x_2^2 - 10.61x_0x_1 - 2.4306x_1x_2 + 2.4182x_1x_2$. 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.

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.

A first step in deriving those equations from the program is to build its control flow graph.

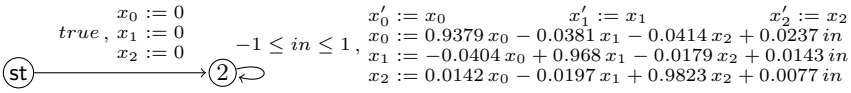

Fig. 2. Control flow graph for our running example.

Example 3. Figure 2 represents the control flow graph for our running example. Vertex "st" corresponds to the starting point of the program and vertex "2" to the head of the loop. The edge between "st" and "2" reflects the three assignments before the loop and the looping edge on vertex "2" represents the loop body.

[4] The word *strategy* is also used in the literature, with equivalent meaning.
[5] This is not illustrated here, one can refer for instance to [17] for more details.
[6] How this template was chosen will be explained later in Section 5.

It is worth noting that, unlike the usual notion of control flow graph with vertices between each single instruction of the program, sequences of instructions are here considered "en bloc" with graph vertices only for starting point and loop heads of the program. This will both improve the precision of the analysis and decrease its cost by avoiding useless intermediate abstractions.

A system of equations is then defined with a variable $b_{i,j}$ for each vertex i of the graph and each template t_j.

Example 4. Here is the system of equations for our running example:

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

where $\mathrm{be}(i)$ is a shortcut for $t_1 \leq b_{i,1} \wedge t_2 \leq b_{i,2} \wedge t_3 \leq b_{i,3} \wedge t_4 \leq b_{i,4}$ and $a(t)$ 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 $\vee$.

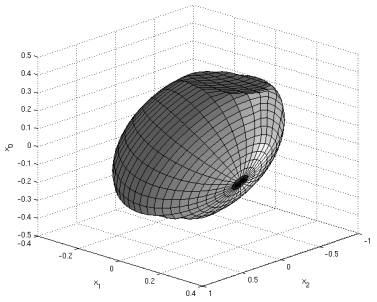

Fig. 3. Invariant for running example.

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: $\max\left\{a(t_j) \;\middle|\; c \wedge \bigwedge_j t_j \leq b_{v,j}\right\}$ where c and a 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 a 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.

Iterating on Policies 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 Iteration [1,19] performs descending iterations towards some fixpoint, working in a way similar to the Newton-Raphson 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.

Max-Policy Iteration [18,19] works in the opposite direction, starting from bottom and iterating computations of greatest fixpoints on so called max-policies until a global fixpoint is reached. The algorithm terminates with a — at least theoretically — precise fixpoint but the user has to wait until the end since intermediate results are not overapproximations of a fixpoint.

In practice, both algorithms compute the same invariant. Min-policies are nonetheless expected to be able to cope with larger systems [19, conclusion].

Example 5. On our running example, policy iterations compute the loop invariant $(1.0029,\ 0.1795,\ 0.1136,\ 0.2757) \in \mathcal{T}$, meaning: $t_1 \leq 1.0029 \wedge t_2 \leq 0.1795 \wedge t_3 \leq 0.1136 \wedge t_4 \leq 0.2757$ or equivalently: $t_1 \leq 1.0029 \wedge |x_0| \leq 0.4236 \wedge |x_1| \leq 0.3371 \wedge |x_2| \leq 0.5251$ (all figures are rounded to the fourth digit). This is a cropped ellipsoid as displayed on Figure 3.

3 An Abstract Control Flow Graph Domain

The previous section stated that policy iterations are able to compute precise fixpoints but require to extract a system of equations from the analyzed program. This fundamentally differs from the classic abstract domain paradigm. Although simply running both kind of analyses in parallel is easy, that would hinder the chances of a tight cooperation between them. The contribution of our work is to define an abstract domain which will gracefully interface both worlds.

This section describes a symbolic abstract domain reconstructing control flow graphs similar to Figure 2. This domain will basically "record" assignments and guards (of if-then-elses conditionals and while loops) in graph edges thanks to the corresponding abstract operators and close loops during widenings.

This will finally be used in the next section to provide an embedding of policy iterations in a classic abstract domain for Kleene iterations.

3.1 Lattice Structure

Definition 1. *Given a set $\mathbb{V}_{ex}$ of additional variables ($\mathbb{V}_{ex} \cap \mathbb{V} = \emptyset$), we define:*
- $\mathcal{A} := \mathbb{V} \to expr$*, functions from variables to expressions on* $\mathbb{V} \cup \mathbb{V}_{ex}$*;*
- $\mathcal{C} := expr \to \overline{\mathbb{R}}$*.*

Variables $\mathbb{V}_{ex}$ will be used for modeling random inputs, $\mathcal{A}$ for *assignments* and $\mathcal{C}$ for *constraints* (mostly coming from guards). We will later write $x := 2y, y := y + 1$ for instance, to denote the function in $\mathcal{A}$ mapping x to the expression $2 \times y$, y to $y + 1$ and every other variable $z \in \mathbb{V}$ to the expression z. Furthermore id will denote the identity function, mapping every variable $x \in \mathbb{V}$ to the expression x. Regarding constraints, $1 \leq x \leq 2 \ \wedge \ y \leq 42$ will represent the function in $\mathcal{C}$ mapping expression x to 2, $-x$ to -1, y to 42 and anything else to $+\infty$. Finally $\bot_{\mathcal{C}}$ is the function in $\mathcal{C}$ mapping every expression to $-\infty$.

Definition 2. *Given a set V, $\mathsf{st} \in V$ and $\mathsf{fi} \in V$ ($\mathsf{fi} \neq \mathsf{st}$) and denoting E the functions $V \times \mathcal{A} \times V \to \mathcal{C}$, we define the set of graphs $\mathcal{G}$ as:*

$$\mathcal{G} := \{ \top_{\mathcal{G}} \} \cup \left\{ (e,t) \in E \times V \;\middle|\; \begin{array}{l} t \neq st \wedge \forall v \in V, \forall a \in \mathcal{A}, e(\mathsf{fi}, a, v) = \bot_{\mathcal{C}} \\ \wedge \forall v \in V, \forall a \in \mathcal{A}, t \neq \mathsf{fi} \Rightarrow e(v, a, \mathsf{fi}) = \bot_{\mathcal{C}} \end{array} \right\}.$$

An element of $\mathcal{G}$ (other than $\top_{\mathcal{G}}$) is a pair (e, t) with e edges of a graph and t a vertex of this graph. t indicates the point of program currently considered by the Kleene iterations, acting like a kind of code pointer on e. The graph e associates to each pair of points v and v' and assignment a the constraint $e(v, a, v')$ to satisfy in order to take this transition and apply assignment a. Among the graph vertices V we distinguish two special vertices: st is the starting point of the program whereas fi will be used as temporary final point. We require two things about fi: that it has no outcoming edge ($\forall v \in V, \forall a \in \mathcal{A}, e(\mathsf{fi}, a, v) = \bot_{\mathcal{C}}$) and that it is used only as current point (if $t \neq \mathsf{fi}$ then fi does not appear in the graph: $\forall v \in V, \forall a \in \mathcal{A}, t \neq \mathsf{fi} \Rightarrow e(v, a, \mathsf{fi}) = \bot_{\mathcal{C}}$).

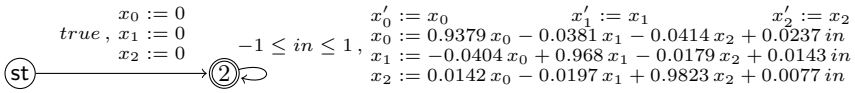

Fig. 4. Example of value in $\mathcal{G}$.

Example 6. For our running example, the result of Kleene iterations at loop head can be represented as on Figure 4. We chose a graphical representation for edges e, drawing only edges different from $\bot_{\mathcal{C}}$, while current point t is represented by a doubly circled vertex. More precisely, we draw an edge labeled (c, a) between v_1 and v_2 when $e(v_1, a, v_2) = c \neq \bot_{\mathcal{C}}$.

An order $\sqsubseteq^\sharp_{\mathcal{G}}$ on $\mathcal{G}$ is basically[7] defined as the pointwise extension on E of $\sqsubseteq^\sharp_{\mathcal{C}}$ (itself a pointwise extension of usual order on $\overline{\mathbb{R}}$) for values with the same current point t. A least upper bound $\bigsqcup^\sharp_{\mathcal{G}}$ is defined likewise, based on the usual max on $\overline{\mathbb{R}}$. This makes $\mathcal{G}$ a complete lattice.

Definition 3 (concretization $\gamma_{\mathcal{G}}$). *Given a template domain $\mathcal{T}$, the concretization function $\gamma_{\mathcal{G}} : \mathcal{G} \to 2^{(\mathbb{V}\to\mathbb{R})}$ is then defined as $\gamma_{\mathcal{G}}(\top_{\mathcal{G}}) = \mathbb{R}^{\mathbb{V}}$ and $\gamma_{\mathcal{G}}(e, t) = \gamma_{\mathcal{T}}(b_{t,1}, \ldots, b_{t,n})$ with $(b_{v,i})_{v\in V, i\in[\![1,n]\!]}$ the least solution of the system of equations previously defined in Section 2.3.*

This concretization function gives a meaning to abstract values. It can be shown to be monotone ($\forall g, g' \in \mathcal{G},\ g \sqsubseteq^\sharp_{\mathcal{G}} g' \Rightarrow \gamma_{\mathcal{G}}(g) \subseteq \gamma_{\mathcal{G}}(g')$). It is also worth noting that, like with any abstract domain, an abstract value $(e, t) \in \mathcal{G}$ overapproximates the reachable state space *at some program point*. The code pointer t is then used to locate this program point in the graph e.

3.2 Abstract Operators

Guards To compute $[\![p \leq r]\!]^\sharp(g)$ for an expression p, a real $r \in \mathbb{R}$ and an abstract value $g \in \mathcal{G}$, we have to distinguish three cases as illustrated on Figure 5:

(a) when $g = \top_{\mathcal{G}}$, typically at starting point, we return a graph with the code pointer at fi and a unique edge between st and fi labeled with $(p \leq r, id)$;
(b) when $g = (e, \mathsf{fi})$, we add the constraint $p \leq r$ to all incoming edges of fi;
(c) finally, when $g = (e, t)$ with $t \neq \mathsf{fi}$. we add an edge labeled with $(p \leq r, id)$ between t and fi.

[7] Up to some details later required for the widening.

Definition 4. *For any expression p, any real number $r \in \mathbb{R}$ and any abstract value $g \in \mathcal{G}$, $[\![p \leq r]\!]^\sharp(g)$ is defined by case analysis on g:*

$[\![p \leq r]\!]^\sharp(\top_{\mathcal{G}}) = (e, \mathsf{fi})$ *where e is the following function:*

$$v, a, v' \mapsto p' \mapsto \begin{cases} r & \text{if } (v, a, v') = (\mathsf{st}, id, \mathsf{fi}), p' = p \\ +\infty & \text{if } (v, a, v') = (\mathsf{st}, id, \mathsf{fi}), p' \neq p \\ -\infty & \text{otherwise} \end{cases};$$

$[\![p \leq r]\!]^\sharp(e, \mathsf{fi}) = (e', \mathsf{fi})$ *where e' is the following function:*

$$v, a, v' \mapsto p' \mapsto \begin{cases} \min\left(r, e(v, a, v')(p')\right) & \text{if } v' = \mathsf{fi}, p' = a(p) \\ e(v, a, v')(p') & \text{otherwise} \end{cases};$$

$[\![p \leq r]\!]^\sharp(e, t) = (e', \mathsf{fi})$ *where e' is the following function:*

$$v, a, v' \mapsto p' \mapsto \begin{cases} r & \text{if } (v, a, v') = (t, id, \mathsf{fi}), p' = p \\ +\infty & \text{if } (v, a, v') = (t, id, \mathsf{fi}), p' \neq p \\ e(v, a, v')(p') & \text{otherwise} \end{cases}.$$

Property 1 (soundness). This abstract operator is sound with respect to the concrete semantics of guards: for all expression p, all real $r \in \mathbb{R}$ and all $g \in \mathcal{G}$, $[\![p \leq r]\!](\gamma_{\mathcal{G}}(g)) \subseteq \gamma_{\mathcal{G}}\left([\![p \leq r]\!]^\sharp(g)\right)$.

Assignments This is very similar to guards, modifying assignments instead of constraints on edges. Soundness property is similar.

Random assignments This is a kind of merge between the two previous ones. The variable randomly assigned in range $[r_1, r_2]$ is first assigned to a fresh new variable[8] $x \in \mathbb{V}_{ex}$ which is then constrained by $-x \leq -r_1$ and $x \leq r_2$. A similar soundness property holds.

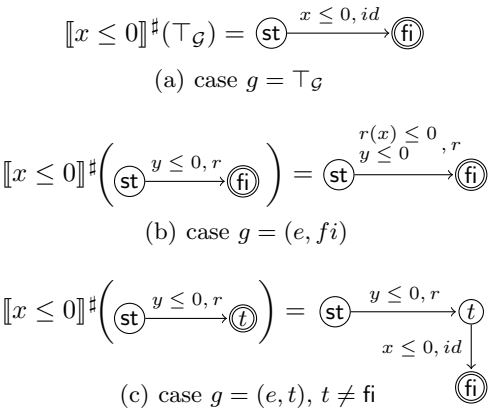

Fig. 5. Examples of abstract guards.

3.3 Widening

On numerical domains, widening discards information in order to enforce termination of the analysis. This is a source of imprecision. On the contrary, the graphs we are computing are finite objects which can be obtained without introducing imprecision. Thus, the following widening operator only aims at closing loops in graphs.

[8] Any variable not appearing in any incoming edge of fi.

None of the abstract operators we have seen up to now introduces new nodes in the graph (other than st and fi). This is done by the widening which plays a key role by introducing new nodes and closing loops on those nodes. Widening is actually the best place to create loops in our abstract control flow graphs since it is usually called at loop heads of programs during analyses[9]. Moreover, in most abstract interpreters, widening is the only indication an abstract domain gets from the presence of loops in the analyzed program.

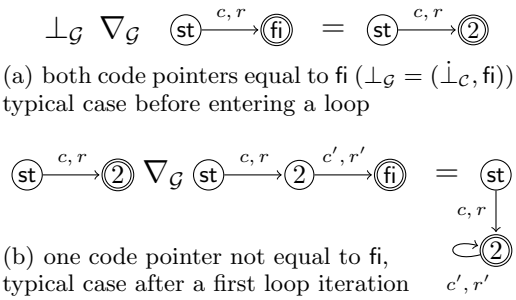

Fig. 6. Widening: introducing new nodes and loops in the control flow graph.

To compute the widening $(e,t)\nabla(e',t')$ of two graphs, there are basically three cases to consider:

- both t and t' are equal to fi: in this case, we create a new point t'' and redirect all incoming edges of fi to t'' in both e and e' before computing their join, this is illustrated on Figure 6 (a);
- either t or t' is not fi (or $t = t' \neq$ fi), say $t \neq$ fi: in this case all incoming edges of fi are redirected to t in e' and a pointwise widening is done on each edge, Figure 6 (b);
- both t and t' are not fi and are different: in this case we return $\top_{\mathcal{G}}$.

4 Embedding Policy Iterations into an Abstract Domain

This section finally describes how to use the control flow graph domain of the previous section to embed policy iterations into a classic abstract domain.

The basic idea is to build a product of the control flow graph domain with a template domain. Policy iterations are then performed during widenings to reduce the template part according to the graph part.

4.1 Reduced product between Graph and Template Domains

First, the template domain $\mathcal{T}$ introduced in Section 2.3 is equipped with dummy abstract operators $[\![.]\!]^\sharp_{\mathcal{T}}$ for guards, assignments and random assignments that always return $\top_{\mathcal{T}} = (+\infty, \ldots, +\infty)$. While perfectly useless, those operators are trivially sound.

Definition 5 (policy iterations domain). *We define the domain $\mathcal{P}_i$ as the product $\mathcal{G} \times \mathcal{T}$ of the domain of previous section with the above template domain.*

This means all operations on $\mathcal{P}_i$ are performed component by component. For instance, for (g,β) and $(g',\beta') \in \mathcal{P}_i$, the join $(g,\beta) \sqcup^\sharp_{\mathcal{P}_i} (g',\beta')$ is defined as[10]

[9] It even *must* be called at least once per loop to ensure convergence of the analyses [4].

[10] The join $\sqcup^\sharp_{\mathcal{T}}$ on $\mathcal{T} = \overline{\mathbb{R}}^n$ is simply the pointwise extension of the usual max on $\overline{\mathbb{R}}$.

$(g \sqcup^\sharp_{\mathcal{G}} g', \beta \sqcup^\sharp_{\mathcal{T}} \beta')$. The concretization function $\gamma_{\mathcal{P}_i} : \mathcal{P}_i \to 2^{\mathbb{V}\to\mathbb{R}}$ is the intersection of the concretizations of each component: $\gamma_{\mathcal{P}_i}(g, \beta) = \gamma_{\mathcal{G}}(g) \cap \gamma_{\mathcal{T}}(\beta)$.

At this point, this domain still looks completely useless. But all the policy iteration work will take place during its widening.

Definition 6 (widening of $\mathcal{P}_i$). *We define* $\nabla_{\mathcal{P}_i} : \mathcal{P}_i \times \mathcal{P}_i \to \mathcal{P}_i$ *as:*

$$(g, \beta) \nabla_{\mathcal{P}_i} (g', \beta') = \begin{cases} (g \nabla_{\mathcal{G}} g', \mathrm{PI}\,(g \nabla_{\mathcal{G}} g')) & \text{if } g' \sqsubseteq^\sharp_{\mathcal{G}} g \\ (g \nabla_{\mathcal{G}} g', \top_{\mathcal{T}}) & \text{otherwise} \end{cases}$$

where $\mathrm{PI}(g)$ *is the result of policy iterations applied on graph*[11] g.

The test $g' \sqsubseteq^\sharp_{\mathcal{G}} g$ is used to perform policy iterations only when the graph domain has stabilized, thus avoiding potentially costly, and rather useless, computations on yet incomplete graphs.

As usual with reduced products [10], $\nabla_{\mathcal{P}_i}$ is not a widening in the strict acceptation of the term, since it is not greater than the join of $\mathcal{P}_i$. However, it still satisfies the two fundamental following properties:

- it does not break the soundness of the analysis since for all $p, p' \in \mathcal{P}_i$: $\gamma_{\mathcal{P}_i}(p \sqcup^\sharp_{\mathcal{P}_i} p') \subseteq \gamma_{\mathcal{P}_i}(p \nabla_{\mathcal{P}_i} p')$;
- it ensures termination of the analysis: for all sequences $x \in {\mathcal{P}_i}^{\mathbb{N}}$, the sequence $y_i = x_i$, $y_{i+1} = y_i \nabla_{\mathcal{P}_i} x_i$ is ultimately stationary.

Equipped with $\nabla_{\mathcal{P}_i}$ as widening operator, $\mathcal{P}_i$ finally offers a classic abstract domain interface to policy iterations.

4.2 Remarks on this Embedding

One may find the previous construction quite complicated and ask why not simply perform policy iterations aside classic Kleene iterations at each loop head. This seemingly simpler approach would however suffer from following drawbacks:

- it is not confined to an abstract domain, breaking the usual abstract interpreter framework [23];
- this would prevent the use of reduced products to exchange information with other domains, since a more static approach would be unable to record those informations on the fly as our graph domain can.

Finally, due to first point, implementation could rapidly become more intricate.

5 Application to Quadratic Invariants on Guarded Linear Systems

Semi-definite programming is a numerical optimization technique allowing by policy iterations to efficiently compute quadratic invariants on linear guarded systems. This short section discusses the interest of such invariants and how to generate adequate templates.

A wide range of today's real-time embedded systems, especially their most critical parts, rely on a control-command computation core. Much, if not most,

[11] More precisely on the system of equations introduced in Section 2.3.

of those systems are based on a linear law (lead-lag, LQR or PID controllers, low-pass filters,...). They periodically update their internal state following a matrix expression of the form $x_{k+1} = Ax_k + Bu_k$ in which x_k represents the state of the system at a given time k, matrix A models the system update according to its previous state while matrix B expresses the effect of the bounded input u_k.

On the one hand, analyzing such systems with linear abstract domains often leads at best to a rather costly analysis or at worst to no result at all. For instance, they often do not admit any invariant in the interval domain. On the other hand, control theorists have known for long that such systems are stable if and only if they admit a quadratic invariant (they call them Lyapunov functions [5]). Those invariants take the shape of an ellipsoid as seen on the running example of Section 2. We demonstrated in previous work [26] how good quadratic templates can be computed by adding appropriate constraints to the previous equation.

Actual programs often contain a number of saturations or resets around the linear core. Those guards are well handled by policy iterations.

6 Experimental Results

All the elements presented in this paper have been implemented as a new abstract domain in our static analyzer for Lustre synchronous programs[12].

For the sake of efficiency, policy iterations are performed with floating point computations using the semidefinite programming library CSDP [2]. This usually yields good results but without any formal guarantee about their correctness[13]. Checking that a result is an actual postfixpoint basically amounts, for each term of the equation system, 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 [27]. Proof of positive definiteness of an $n \times n$ matrix can then be achieved with $\mathrm{O}(n^3)$ floating point operations, which in practice induces only a very small overhead to the whole analysis.

Experiments were conducted on a set of stable linear systems. These systems were extracted from [1,13,26]. We have to recall to the reader that those systems, despite their apparent simplicity, do not admit simple linear invariants. Table 1 sums up analysis times for various versions of them, with or without saturations or resets. All computations were performed on an Intel Core2 @ 2.66GHz. It is interesting to notice that we nearly always get better results than [1,26] either thanks to the better templates obtained by solving Lyapunov equations (compared to [1]) or thanks to the extra templates bounding each variable (compared to [26]). Moreover, those quadratic invariants are fully automatically inferred from the analyzed program, while [1] requires the user to supply them. Although [12] may infer better bounds for the first two examples thanks to a kind of unrolling mechanism, Fluctuat [20] and its zonotopes is, to the best of authors' knowledge, the only abstract interpreter that may be able to automatically bound the other examples. This would however be a lot more expensive.

[12] Because we had it at hand. This only advocates the versatility of the approach.

[13] Again, we speak here about the soundness of the result (the fixpoint computed) w.r.t. the real semantics of the program and not its floating point one.

Table 1. Result of the experiments: quadratic invariants inference. Column n gives the number of program variables considered for policy iteration. The remaining columns detail the computation time: *templates* corresponds to the quadratic template computation, *iterations* to the actual policy iterations and *check* to the soundness checking. For each example, except the last one, the first line is for the bare linear system, the second for the same system with an added saturation and the third with a reset. $\bot$ indicates failure of the soundness checking.

	n	total (s)	templates (s)	iterations (s)	check (s)
Ex. 1 From [13, slides]	3	0.47	0.38	0.05	0.01
	4	1.26	0.70	0.37	$\bot$ (0.00)
	3	0.56	0.41	0.09	0.02
Ex. 2 From [13, slides]	5	0.70	0.56	0.05	0.02
	6	1.18	0.57	0.37	0.12
	5	0.82	0.59	0.10	0.04
Ex. 3 Discretized lead-lag controller	3	0.53	0.35	0.13	0.02
	4	1.06	0.36	0.54	0.08
	3	0.64	0.35	0.23	0.03
Ex. 4 Linear quadratic gaussian regulator	4	0.66	0.38	0.19	0.03
	5	1.33	0.39	0.63	0.14
	4	0.90	0.38	0.38	0.06
Ex. 5 Observer based controller for a coupled mass system	6	1.12	0.66	0.24	0.06
	7	2.59	0.65	1.34	0.26
	6	1.39	0.67	0.42	0.11
Ex. 6 Butterworth low-pass filter	6	1.39	0.99	0.17	0.07
	7	2.64	1.01	1.05	0.22
	6	1.63	1.00	0.31	0.12
Ex. 7 Dampened oscillator from [1]	2	0.35	0.21	0.07	0.01
	3	1.25	0.24	0.28	0.09
	2	0.44	0.23	0.14	0.03
Ex. 8 Harmonic oscillator from [1]	2	0.36	0.22	0.07	0.01
	3	0.82	0.20	0.44	0.10
	2	0.44	0.22	0.13	0.03
Ex. 5 and 6 chained	$6+6$	2.53	$0.67+1.00$	$0.24+0.17$	$0.06+0.06$
	12	7.92	4.06	2.00	0.52

The analyzer is released under a GPL license and available along with all examples and results at `http://cavale.enseeiht.fr/policy2013/`.

Example 7. The last line of Table 1 considers two linear systems chained, the output of the first one being used as input by the second one. This program is first analyzed with two policy iteration domains communicating together via reduced product to and from the intervals domain (the two domains do not share any variable). It is worth noting that total analyses time is just the sum of the times needed for the two separate analyses. In comparison, the second analysis with one single domain for the whole program is much more expensive.

7 Related Work

Multiple approaches try to tackle the loss of precision of Kleene iterations. A first line of work concerns recent developments to improve widening [21] and

narrowing [22]. However those approaches cannot guarantee to reach the least fixpoint. Furthermore the authors are not aware of any numerical domain able to compute quadratic invariants based on such advanced widening. We recall that the examples presented in the Section 6 do not admit any simple inductive linear invariant.

Policy iteration techniques are another approach. We address the interested reader – beyond our policy iteration primer of Section 2 – to the seminal papers using semidefinite programming: [1,19] for the Min-Policy and [18,19] for the Max-Policy. All those works require appropriate templates for the use of policy iteration, while our instantiation of the framework is fully automatic. Furthermore, they make use of floating point semidefinite programming, but without addressing the soundness issue as we do. They do however acknowledge this fact.

About the integration of policy iterations and classic abstract interpretation, the opposite approach of the current paper has been proposed in [30]. The authors introduced additional transformers in order to extend a numerical abstract domain to its use with policy iterations. Due to this modification of the abstract domain interface, it does not give an embedding of policy iterations in a classic abstract domain as offered in the current paper.

We should also mention alternatives to classic widening, other than policy iterations. These are called acceleration techniques [3,11,28]. They compete with policy iterations but hardly extend to non linear properties.

About the analysis of guarded linear systems, the work [12,13,25] addresses a strict subclass of the programs handled by our tool. However since they rely on some kind of unrolling, they could be more precise for such specific problems. Maximum reachable values (our bounds are usually a few percents larger) can be computed via support functions [29]. However, due to heavy unrolling, only pure linear systems, without guards, are handled and the result is not an inductive invariant.

The generation of quadratic ellipsoid templates was already presented in [26] but this paper did not make use of policy iterations and the approach was only applicable to models of linear systems without if-then-else statements, not on actual program sources.

Last, as already mentioned at the end of Section 3, the work [17] relies on an SMT solver to optimize the policy choice when computing Max-policy iterations. In fact an important system with multiple if-then-else construct will lead to an exponential number of policies. Having an implicit representation and a means to make an efficient choice is then essential. Although this work has only been applied for linear templates, its extension to our framework should be of interest.

8 Conclusion and Future Work

To the author's knowledge this paper presents the *first integration of policy iteration as a fully usable relational abstract domain*. This integration in a Kleene fixpoint is enabled thanks to (i) an *abstract domain* that *rebuilds the control flow graph* and allows the policy iteration algorithm to access a global view of the program as a system of equations; (ii) a method, based on [26], to *synthetize meaningful ellipsoid templates* for a specific class of programs: stable guarded linear systems. This provides a powerful abstract domain able to compute non

linear invariants in a fully automatic way, in a manner similar to relational abstractions such as polyhedra.

Reduction between classic domains and our allows both to precisely represent this control flow graph and to inject the result of policy iterations within classic domains. It also enables the use of multiple policy iteration domains; for example when considering sequences of such guarded linear filters as in Example 7.

The experimental results showed that this approach really extends the applicability of Kleene-based abstract interperter to a wider class of systems admitting non linear invariants. When computing our analyzes we only provided the set of variables that have to be analyzed with policy iterations, without any other information like templates.

Finally the issue of floating point semantics should not be forgotten. The introduction of error terms has to be addressed.

Acknowledgments. We deeply thank Éric Goubault, Peter Schrammel and the anonymous reviewers for useful comments regarding this paper.

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. Brian Borchers. Csdp, a c library for semidefinite programming. *Optimization Methods and Software*, 11(1-4), 1999.
3. Olivier Bouissou, Yassamine Seladji, and Alexandre Chapoutot. Acceleration of the abstract fixpoint computation in numerical program analysis. *J. Symb. Comput.*, 47(12), 2012.
4. François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications*. 1993.
5. 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.
6. 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.
7. Patrick Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI*, 2005.
8. 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.
9. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.
10. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the Astrée static analyzer. In *ASIAN*, 2006.
11. Paul Feautrier and Laure Gonnord. Accelerated invariant generation for c programs with aspic and c2fsm. *Electr. Notes Theor. Comput. Sci.*, 267(2), 2010.
12. Jérôme Feret. Static analysis of digital filters. In *ESOP*, number 2986, 2004.

13. Jérôme Feret. Numerical abstract domains for digital filters. In *International workshop on Numerical and Symbolic Abstract Domains (NSAD)*, 2005.
14. Stephane Gaubert, Eric Goubault, Ankur Taly, and Sarah Zennou. Static analysis by policy iteration on relational domains. In *ESOP*, 2007.
15. Thomas Gawlitza and Helmut Seidl. Precise fixpoint computation through strategy iteration. In *ESOP*, 2007.
16. Thomas Gawlitza and Helmut Seidl. Precise relational invariants through strategy iteration. In *CSL*, 2007.
17. Thomas Martin Gawlitza and David Monniaux. Improving strategies via smt solving. In *ESOP*, 2011.
18. Thomas Martin Gawlitza and Helmut Seidl. Computing relaxed abstract semantics w.r.t. quadratic zones precisely. In *SAS*, 2010.
19. Thomas Martin Gawlitza, Helmut Seidl, Assalé Adjé, Stéphane Gaubert, and Eric Goubault. Abstract interpretation meets convex optimization. *J. Symb. Comput.*, 47(12), 2012.
20. Khalil Ghorbal, Eric Goubault, and Sylvie Putot. The zonotope abstract domain taylor1+. In *CAV*, 2009.
21. Denis Gopan and Thomas W. Reps. Lookahead widening. In *CAV*, volume 4144 of *Lecture Notes in Computer Science*. Springer, 2006.
22. Nicolas Halbwachs and Julien Henry. When the decreasing sequence fails. In *SAS*, volume 7460 of *Lecture Notes in Computer Science*. Springer, 2012.
23. Bertrand Jeannet. Some experience on the software engineering of abstract interpretation tools. *Electr. Notes Theor. Comput. Sci.*, (2), 2010.
24. Antoine Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, October 2001.
25. David Monniaux. Compositional analysis of floating-point linear numerical filters. In *CAV*, 2005.
26. Pierre Roux, Romain Jobredeaux, Pierre-Loïc Garoche, and Éric Féron. A generic ellipsoid abstract domain for linear time invariant systems. In *HSCC*. ACM, 2012.
27. Siegfried M. Rump. Verification of positive definiteness. *BIT Numerical Mathematics*, 46, 2006.
28. Peter Schrammel and Bertrand Jeannet. Logico-numerical abstract acceleration and application to the verification of data-flow programs. In *SAS*, 2011.
29. Yassamine Seladji and Olivier Bouissou. Numerical abstract domain using support functions. In *NFM*, 2013.
30. Pascal Sotin, Bertrand Jeannet, Franck Védrine, and Eric Goubault. Policy iteration within logico-numerical abstract domains. In *ATVA*, 2011.