Abstract Fixpoint Computations with Numerical Acceleration Methods

Olivier Bouissou[1], Yassamine Seladji[1]

CEA, LIST, Laboratory for the Modeling and Analysis of Interacting Systems
91191 Gif-sur-Yvette, France

Alexandre Chapoutot[2]

Université Pierre et Marie Curie, LIP6 – Paris, France

Abstract

Static analysis by abstract interpretation aims at automatically proving properties of computer programs. Basically, an over-approximation of program semantics, defined as the least fixpoint of a system of semantic equations, must be computed. To enforce the convergence of this computation, widening operator is used but it may lead to coarse results. We propose a new method to accelerate the computation of this fixpoint by using standard techniques of numerical analysis. Our goal is to automatically and dynamically adapt the widening operator in order to maintain precision.

Keywords: Abstract numerical domains, acceleration of convergence, widening operator.

1 Introduction

In the field of static analysis of embedded numerical programs, abstract interpretation [**?**,**?**] is widely used to compute over-approximations of the set of behaviors of programs. This set is usually defined as the least fixpoint of a monotone map on an abstract domain given by the (abstract) semantics of the program. Using Tarski's theorem [**?**], this fixpoint is computed as the limit of the iterates of an abstract function starting from the least element. These iterates build a sequence of abstract elements that (order theoretically) converges towards the least fixpoint. Since this sequence may converge slowly (or only

[1] Email: olivier.bouissou@cea.fr, yassamine.seladji@cea.fr
[2] Email: alexandre.chapoutot@lip6.fr

after infinitely many steps), the theory of abstract interpretation introduces the concept of *widening* [**?**].

A widening operator is a two-argument function ∇ which tries to predict the limit of the iterates based on the relative position of two consecutive iterates. For example, the standard widening operator on the interval abstract domain consists in comparing the limits of the intervals and setting the unstable ones to ∞ (or $-\infty$). A widening operator often makes large over-approximation because it must make the sequence of iterates converge in a finite time. Over-approximation may be reduced afterward using a *narrowing* operator but the precision of the final approximation still strongly depends on the precision of ∇. Various techniques have been proposed to improve it. *Delayed* widening makes use of ∇ after n iteration steps only (where n is a user-defined integer), thus letting the first loop iterates execute before trying to predict the limit. The delay parameter n usually has to be defined a priori. Another approach is to use a widening with *thresholds* [**?**]: the upper bound of the interval (for example) is not directly set to ∞, but is successively increased using a set of thresholds that are candidates for the value of the fixpoint upper bound. In practice, these techniques are necessary to obtain precise fixpoint approximations for industrial-sized embedded programs. However, they suffer from their lack of automatization: thresholds must be chosen *a priori* and are defined by the user. Some methods try to automatically discover thresholds from the program [**?**,**?**]: whenever an inequality (e.g. the condition of a loop) is found, a threshold (or landmark in [**?**]) is added, its value depending on the constants appearing in the inequality. So the thresholds are based on a *syntactic* criterion; in our work we define thresholds using the *dynamics* of the program variables. As a consequence, the use of a static analyzer is difficult as these (non-trivial) parameters are often hard to find.

In this article, we present some ongoing work which shows that it is possible to use *sequence transformation techniques* in order to automatically and efficiently derive an approximation to the limit of Kleene iterates. This approximation may not be safe (i.e. may not contain the actual limit), but we show how to use it in the sense of abstract interpretation. Sequence transformation techniques (also known as convergence acceleration methods) are widely studied in the field of numerical analysis [**?**]. They transform a converging sequence $(x_n)_{n\in\mathbb{N}}$ of real numbers into a new sequence $(y_n)_{n\in\mathbb{N}}$ which converges faster to the same limit (see Section 3.2). In some cases (depending on the method), the acceleration is such that $(y_n)_{n\in\mathbb{N}}$ is ultimately constant. Some recent work [**?**] applied these techniques in the case of sequences of *vectors* of real numbers: vector sequence transformations introduce *relations* between elements of the vector and perform better than scalar ones. Our main contribution is to show that we can use these methods in order to improve the fixpoint computation in static analysis: we define *dynamic* thresholds for

widening that are very close to the actual fixpoint. This increased precision is obtained because sequence transformations use all iterates and *quantitative information* (i.e. relative to the distance between elements) to predict the limit. They thus exploit more information than the widening and make a better prediction. In this work, we focus on the interval domain, but we believe that this work may be applied to any abstract domain, especially the ones with a *pre-defined shape* (octagons [**?**], templates [**?**], etc.) =¿ Insister sur le fait qu'on utilise le domaine numérique des intervalles pour expliquer notre technique, puis nous l'appliquerons en utilisant le domaine abstrait numérique des octogones. Let us remark that our techniques are well-suited for accelerating the invariant generation of numerical programs with floating-point variables and that we do not address the case of integer variables as in [**?**,**?**].

This article is organized as follows. In Section 2, we explain on a simple example how acceleration methods may be used to speed-up the fixpoint computation. In Section 3, we recall the theoretical basis of this work and present our main theoretical contribution in Section 4. Section 5 presents some early experiments on various floating-point programs that show the interest of our approach, while Sections 6 and 7 discuss related works and perspectives.

Notations. In the rest of this article, (x_n) will denote a sequence of real numbers (i.e. $(x_n) \in \mathbb{R}^{\mathbb{N}}$), while $(\boldsymbol{x}_n)$ denotes a sequence of vector of real numbers (i.e. $(\boldsymbol{x}_n) \in \left(\mathbb{R}^p\right)^{\mathbb{N}}$ for some $p \in \mathbb{N}$). The symbol X_n will be used to represent *abstract iterates*, i.e. $X_n \in A$ for some abstract lattice A.

2 An introductive example

Étaler un peu plus cette partie car je trouve que les informations sont condensées In this section, we explain, using a simple example, how sequence acceleration techniques can be used in the context of static analysis. In short, our method works as follows: let (X_n) be a sequence of intervals computed by the Kleene iteration and that is chosen to be widened (see [**?**] for details on how to choose the widening points). From (X_n) we extract a vector sequence $(\boldsymbol{x}_n)$: at stage k, $\boldsymbol{x}_k$ is a vector that contains the infimum and supremum of each variable of the program. As Kleene iteration converges towards the least fixpoint of the abstract transfer function, the sequence $(\boldsymbol{x}_n)$ converges towards a limit $\boldsymbol{x}$ which is the vector containing the infimum and the supremum of this fixpoint. We then compute an accelerated sequence $(\boldsymbol{y}_n)$ that converges towards $\boldsymbol{x}$ faster than $(\boldsymbol{x}_n)$. Once this sequence has reached its limit (or is sufficiently close to it), we use $\boldsymbol{x}$ as a threshold for a widening on $(\boldsymbol{x}_n)$ and thus obtain, in a few steps, the least fixpoint. In the rest of this section, we detail these steps.

```
1  while (1) {
2    xn1 = −0.4375 * x1+ 0.0625 * x2 + 0.2652 * x3 + 0.1 * u1;
3    xn2 = 0.0625 * x1 + 0.4375 * x2 + 0.2652 * x3 + 0.1 * u2;
4    xn3 = −0.2652 * x1 + 0.2652 * x2 + 0.375 * x3 + 0.1 * u3;
5    x1 = xn1; x2 = xn2; x3 = xn3;
6   }
```

Fig. 1. A simple linear program.

The program. We consider a linear program which iterates the function $F(X) = A \cdot X + B \cdot U$ where A, B and U are constant matrices and X is the vector of variables (see Figure 1). Initially, we have $\texttt{x1} \in [1, 2]$, $\texttt{x2} \in [1, 4]$, $\texttt{x3} \in [1, 20]$, $\texttt{u1} \in [1, 6]$, $\texttt{u2} \in [1, 4]$ and $\texttt{u3} \in [1, 2]$. Using an interval analysis, we showed that this program converges in 55 iterations (without widening) and obtained the invariant $[-5.1975, 8.8733]$ for `x1` at line 2.

Extracting the sequence. From this program, we can define a vector sequence of size 6, $\boldsymbol{x}_n = \left(\underline{x^1_n}, \overline{x^1_n}, \underline{x^2_n}, \overline{x^2_n}, \underline{x^3_n}, \overline{x^3_n}\right)$, which represents the evolution of the supremum and the infimum of each variable `x1`, `x2` and `x3` at line 2. For example, the sequence $(\overline{x^1_n})$ is recursively defined by:

$$\overline{x^1_{n+1}} = \max\left(\overline{x^1_n} \ , \ -0.4375 * \underline{x^1_n} + 0.0625 * \overline{x^2_n} + 0.2652 * \overline{x^3_n} + 0.1 * \overline{u_1} \right) . \tag{1}$$

Note that we are not interested in the formal definition of these sequences (as given by Equation (1)), but only in their numerical values that are extracted from Kleene iterates. Each sequence $(\overline{x^i_n})$ (resp. $(\underline{x^i_n})$) is increasing (resp. decreasing) and the sequence $(\boldsymbol{x}_n)$ converges towards a vector $\boldsymbol{x}$ containing the infima and the suprema of the fixpoint (see Figure 2, dotted lines).

Accelerating the sequence. We then used the *vector ε-algorithm* [**?**] to build a new sequence that converges faster towards $\boldsymbol{x}$. This method works as follows (a more formal definition will be given in Section 3.2): it computes a series of sequences $(\boldsymbol{\varepsilon}^k_n)$ for $k = 1, 2, \ldots$ such that each sequence $(\boldsymbol{\varepsilon}^k_n)$ for k *even* converges towards $\boldsymbol{s}$ and the *diagonal* $(\boldsymbol{d}_n) = (\boldsymbol{\varepsilon}^{2n}_0)$ also converges towards $\boldsymbol{s}$. This diagonal sequence is the result of the ε-algorithm and it is called the *accelerated sequence*. It converges faster than the original sequence: in only 8 iterates (which require 16 iterates of the original sequence, as will be explained later), it reached the fixpoint and stayed constant (see Figure 2, bold lines).

Using the accelerated sequence. When the accelerated sequence reaches the limit (or is sufficiently close to it), we modify the Kleene iteration and directly jump to the limit. Formally, if the limit is $(\underline{x_1}, \overline{x_1}, \underline{x_2}, \overline{x_2}, \underline{x_3}, \overline{x_3})$ and if the current Kleene iterate is X_p, we construct the abstract element X whose bounds are $\underline{x_1}$, $\overline{x_1}, \ldots$ and set $X_{p+1} = X_p \cup X$ and re-start Kleene iteration from X_{p+1}. In this way, we remain sound ($X_p \subseteq X_{p+1}$) and we are very close to the fixpoint, as $X \subseteq X_{p+1}$. In this example, Kleene iteration stopped after 2 steps and reached the same fixpoint as the one obtained

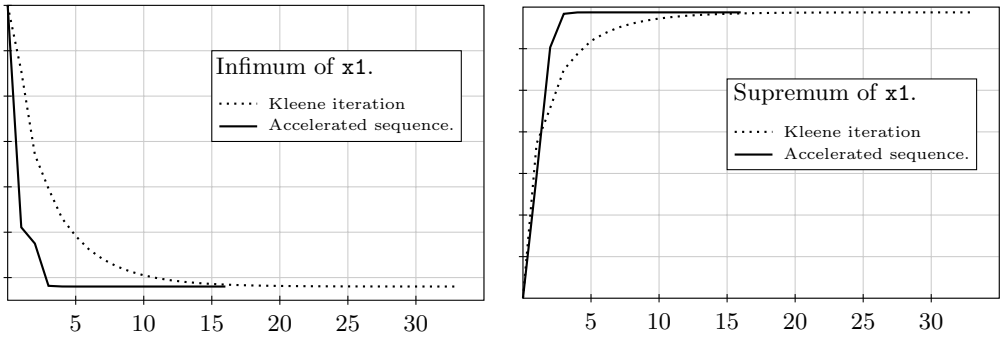


Fig. 2. Sequences extracted from the program of Figure 1 and their accelerated version.

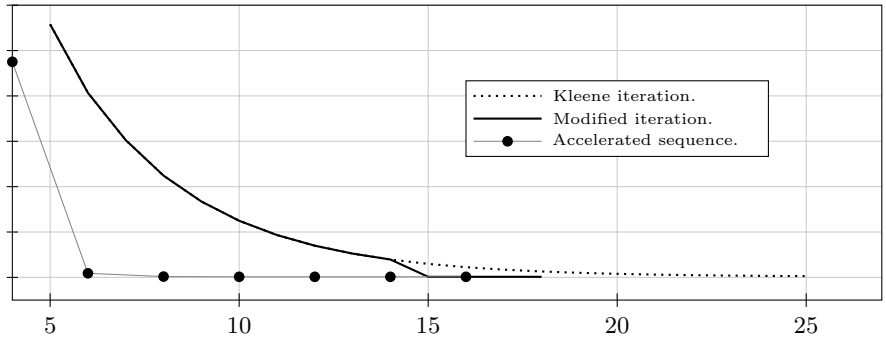


Fig. 3. Infimum value of `x1`. We only display the iterates 5 to 25. At the 15th iteration, the accelerated value is used as a widening with thresholds, and the iteration stops after 18 steps.

without widening and acceleration. Figure 3 shows the original Kleene iteration and the modified one, for the infimum of variable `x1`. Let us recall that the Kleene iteration needed 55 steps to converge, where the modified iteration stops after 18 steps.

3 Theoretical frameworks

In this section, we briefly recall the basics of abstract interpretation, with an emphasis on the widening operator. Next, we present the theory of sequence transformations in more details.

3.1 Overview of abstract interpretation theory

Abstract interpretation is a general method to compute over-approximations of program semantics defined by a monotone semantic function F. The two key ideas are:

- *Safe abstractions* of sets of states based on, in the more general framework [**?**, Sect. 7], concretization functions. More precisely let $\langle C, \sqsubseteq_C \rangle$ be the lattice of concrete states and let $\langle A, \sqsubseteq_A \rangle$ be the lattice of abstract states. The concretization function is a monotonic map $\gamma : A \to C$. We consider

$x \in A$ as a safe abstraction of $y \in C$ if $y \sqsubseteq_C \gamma(x)$.
Moreover, the abstract monotone semantic function $F^\sharp$ is a safe abstraction of F iff $\forall x \in A, F(\gamma(x)) \sqsubseteq_C \gamma(F^\sharp(x))$. Expliquer cette partie en donnant et utilisant l'algorithme de Kleene, afin de pouvoir dire par la suite que notre technique améliore le calcul du point fixe sans trop changer l'algorithme de kleene.

- An *effective* computation method using a widening operator when abstract semantics are based on infinite height lattices. The abstract program semantics is a set of states X of a lattice $\langle A, \sqsubseteq_A \rangle$ such that $X = F^\sharp(X)$. The solution X is iteratively constructed by $X_{i+1} = X_i \sqcup F^\sharp(X_i)$, starting from $X_0 = \bot$. The value $\bot$ and the operation $\sqcup$ denote the smallest element and the join operation of A respectively. The sequence (X_n) defines an increasing chain of elements of A. This chain may be infinite, so to enforce the convergence of this sequence, we substitute the operator $\sqcup$ by a *widening operator* ∇, see Definition 3.1, that is an over-approximation of $\sqcup$.

Definition 3.1 (Widening operator [?]) *Let $\langle A, \sqsubseteq_A \rangle$ be a lattice. The map $\nabla : A \times A \to A$ is a widening operator iff i) $\forall v_1, v_2 \in A$, $v_1 \sqcup v_2 \sqsubseteq_A v_1 \nabla v_2$. ii) For each increasing chain $v_0 \sqsubseteq_A \cdots \sqsubseteq_A v_n \sqsubseteq_A \cdots$ of A, the increasing chain defined by $s_0 = v_0$ and $s_n = s_{n-1} \nabla v_n$ is stationary: $\exists n_0, \forall n_1, n_2, (n_2 > n_1 > n_0) \Rightarrow s_{n_1} = s_{n_2}$.*

The widening operator plays an important role in static analysis because it allows to consider infinite state spaces where the ascending chain condition is not satisfied. Many abstract domains are thus associated with a widening operator; for the interval domain, for example, it is usually defined by:

$$[a,b] \nabla [c,d] = \left[\begin{cases} a & \text{if } a \leq c \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ +\infty & \text{otherwise} \end{cases} \right].$$

Note that we only consider two consecutive elements to extrapolate the potential fixpoint. The main drawback with this widening is that it may generate too coarse results by quickly going to infinity. A solution of this is to add intermediate steps among a finite set T; that is the idea behind the *widening with thresholds* ∇_T. For the interval domain, it is defined [?] by: Je pense qu'il ne faut pas mettre le même ensemble T pour les bornes sup et inf. Que pensez-vous?

$$[a,b] \nabla_T [c,d] = \left[\begin{cases} a & \text{if } a \leq c \\ \max\{t \in T : t \leq c\} & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ \min\{t \in T : t \geq d\} & \text{otherwise} \end{cases} \right].$$

While widening with thresholds gives better results, we are facing with the problem to define *a priori* the set T. Finding relevant values for T is a difficult task for which only syntactic-based techniques exist [?,?].

3.2 Acceleration of convergence

We give an overview of the techniques of acceleration of convergence in numerical analysis (for more details, we refer to [?]). The goal of convergence

acceleration techniques, also named *sequence transformations*, is to increase the rate of convergence of a sequence. Formally, let $\big(D, d\big)$ be a metric space, i.e. a set D with a distance $d : D \to \mathbb{R}^+$ (D will be $\mathbb{R}$ or $\mathbb{R}^p$ for some $p \in \mathbb{N}$). The set of sequences over D (denoted $D^{\mathbb{N}}$) is the set of functions between $\mathbb{N}$ and D. A sequence $(x_n) \in D^{\mathbb{N}}$ converges to ℓ iff we have $\lim_{n\to\infty} d(x_n, \ell) = 0$. A *sequence transformation* is a function $T : D^{\mathbb{N}} \to D^{\mathbb{N}}$ (T designs a particular acceleration method) such that whenever (x_n) converges to ℓ then $(y_n) = T(x_n)$ also converges to ℓ and $\lim_{n\to\infty} \frac{d(y_n,\ell)}{d(x_n,\ell)} = 0$. This means that (y_n) is asymptotically closer to ℓ than (x_n). An important notion for a sequence transformation T is its kernel K_T which is the set of sequences (x_n) for which $T(x_n)$ is ultimately constant. We now present some acceleration methods that we used in our experimentation.

3.2.1 The Aitken Δ^2-method

It is probably the most famous sequence transformation. Given a sequence $(x_n) \in \mathbb{R}^{\mathbb{N}}$, the accelerated sequence (y_n) is defined by: $\forall n \in \mathbb{N},\ y_n = x_n - \frac{x_{n+1}-x_n}{x_{n+2}-2x_{n+1}+x_n}$. It should be noted that in order to compute y_n for some $n \in \mathbb{N}$, three values of (x_n) are required: x_n, x_{n+1} and x_{n+2}. The kernel K_{Δ^2} of this method is the set of all sequences of the form $x_n = s + a.\lambda^n$ where s, a and λ are real constants such that $a \neq 0$ and $\lambda \neq 1$ (see [**?**]). The Aitken Δ^2-method is an efficient method for accelerating sequences, but it highly suffers from numerical instabilities when x_n, x_{n+1} and x_{n+2} are close to each other.

3.2.2 The ε-algorithm

It is often cited as the best general purpose sequence transformation for slowly converging sequences [**?**]. From a converging sequence $(x_n) \in \mathbb{R}^{\mathbb{N}}$ with limit ℓ, the ε-algorithm builds the following sequences:

$$(\varepsilon_n^{-1}) : \forall n \in \mathbb{N}, \varepsilon_n^{-1} = 0, \tag{2}$$
$$(\varepsilon_n^{0}) : \forall n \in \mathbb{N}, \varepsilon_n^{0} = x_n, \tag{3}$$
$$(\varepsilon_n^{k}) : \forall k \geq 1,\ \forall n \in \mathbb{N},\ \varepsilon_n^{k+1} = \varepsilon_{n+1}^{k-1} + \left(\varepsilon_{n+1}^{k} - \varepsilon_n^{k}\right)^{-1} \tag{4}$$

For a fixed k, the sequence $((\varepsilon_n^k)_{n\in\mathbb{N}})$ is called the k-th column, and its construction can be graphically represented as on Figure 4. The *even* columns (ε_n^{2k}) (in gray on Figure 4) converge faster to ℓ. The *even* diagonals $((\varepsilon_n^{2k})_{k\in\mathbb{N}})$ also converges faster to ℓ. In particular, the first diagonal (circled in Figure 4) converges very quickly to ℓ, and it is the accelerated sequence. Let us remark that in order to compute the p-th element of that sequence, $(2p-1)$ elements of (x_n) are required, as stated by Proposition 3.2.

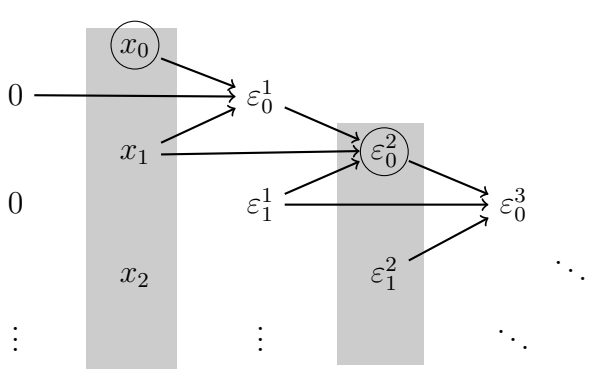
Proposition 3.2 *Let $((S_I)_n)$ be a sequence and let $((S_A)_n) = (\varepsilon_0^{2n})$ its accelerated version given by the ε-algorithm. Then the p-th element of $((S_A)_n)$ is*

defined by $2p-1$ *elements of* $((S_I)_n)$.

Proof. We call $G(n,k)$ the number of elements from $((S_I)_n)$ required to compute the element ε_n^k of indices n and k in the ε-algorithm. By construction, we have: $\forall k \geq 1,\ \forall n \in \mathbb{N},\ \varepsilon_n^{k+1} = \varepsilon_{n+1}^{k-1} + \left(\varepsilon_{n+1}^k - \varepsilon_n^k\right)^{-1}$ so $G(n,k+1) = \max\left(G(n+1,k-1), G(n+1,k), G(n,k)\right)$. Note that the function G is increasing in n and k, so $G(n,k+1) = G(n+1,k)$. Thus, we define the function G by: $G(n,k) = \begin{cases} n+1 & \text{if } k=0 \\ G(n+1,k-1) & \text{otherwise} \end{cases}$. Moreover following the ε-algorithm, we know that $(S_A)_p$, the element of index p in $((S_A)_n)$, i.e. the $(p+1)$-th element, is ε_0^{2p}. We easily prove by recurrence that $\forall p \in \mathbb{N},\ G(0,2p) = 2p+1$.

$$\begin{aligned} G(0,2p) &= G(1,2p-1) \\ &= G(1+2p-1,0) \quad \text{(By recurrence on: G(n,k)=G(n+k,0))} \\ &= G(2p,0) = 2p+1 \end{aligned}$$

So, to have the element $(S_A)_p$, we need $(2p+1)$ elements from $((S_I)_n)$. We know that the element $(S_A)_p$ is the $(p+1)$-th element of $((S_A)_n)$. So, to obtain p elements of $((S_A)_n)$, $2p-1$ elements of $((S_I)_n)$ are required. □



Arrows depict dependencies: the element at the beginning of the arrow is required to compute the element at the end. For example, the second element of the accelerated sequence is:

$$\begin{aligned} \varepsilon_0^2 &= \varepsilon_1^0 + \frac{1}{\varepsilon_1^1 - \varepsilon_0^1} \\ &= x_1 + \frac{1}{\varepsilon_2^{-1} + \frac{1}{\varepsilon_2^0 - \varepsilon_1^0} - \varepsilon_1^{-1} + \frac{1}{\varepsilon_1^0 - \varepsilon_0^0}} \\ &= x_1 + \frac{1}{\frac{1}{x_2 - x_1} - \frac{1}{x_1 - x_0}} \end{aligned}$$

Fig. 4. The ε-table

3.2.3 Acceleration of vector sequences

Many acceleration methods were designed to handle scalar sequences of real numbers. For almost each of these methods, extensions have been proposed to handle vector sequences (see [**?**] for a review of them). The simplest, yet one of the most powerful, of these methods is the *vector ε-algorithm* (VEA). Note that, in this article, we only consider VEA for the acceleration method of vector sequences. Given a vector sequence $(\boldsymbol{x}_n)$, the VEA computes a series of vector sequences $(\boldsymbol{\varepsilon}_n^k)$ using Equations (2)-(4) where the arithmetic operations $+$ and $-$ are computed component-wise and the inverse of a vector $\boldsymbol{v}$ is computed as $\boldsymbol{v}^{-1} = \boldsymbol{v}/(\boldsymbol{v}\cdot\boldsymbol{v})$, with / being the component-wise division

and $\cdot$ the scalar product. The VEA differs from a component-wise application of the (scalar) ε-algorithm as it introduces *relations* between the components of the vector: the scalar product $\boldsymbol{v} \cdot \boldsymbol{v}$ computes a global information on the vector $\boldsymbol{v}$ which is propagated to all components. Our experiments show that this algorithm works better than a component-wise application of the ε-algorithm. The kernel K_ε of the VEA contains all sequences of the form $\boldsymbol{x}_{n+1} = A\boldsymbol{x}_n + B$, where A is a constant matrix and B a constant vector [**?**].

4 Accelerated Kleene iteration

In this section, we combine acceleration methods with the abstract fixpoint computation. Our goal is to be as non-intrusive as possible in the classical iterative scheme. In this way, our method can be implemented with minor adaptations in current static analyzers.

4.1 Methodology

As seen in Section 3.1, the Kleene iteration for finding the least fixpoint is based on abstract values from some abstract lattice A. In order to use acceleration techniques on the abstract iterates, we need to extract a vector of real numbers from the abstract elements $X_n \in A$. We obtain a sequence of real vectors that we can accelerate, and we quickly reach its limit. We then construct an abstract element X that corresponds to this limit and use it as a candidate for the least fixpoint. This process of transforming an abstract value into a real vector and back is formalized by the notion of *extraction* and *combination* functions that are given in Definition 4.1.

Definition 4.1 [Extraction and combination.] Let $\langle A, \sqsubseteq_A \rangle$ be an abstract domain, and let $p \in \mathbb{N}$. The functions $\Lambda_A : A \to \mathbb{R}^p$ and $\Upsilon_A : \mathbb{R}^p \to A$ are called extraction and combination function, respectively, iff for each sequence $X_n \in A^{\mathbb{N}}$ that *order theoretically converges*, i.e. $\sqcup_{n\in\mathbb{N}} X_n = X$ for some $X \in A$, then the sequence $\Lambda_A(X_n) \in \left(\mathbb{R}^p\right)^{\mathbb{N}}$ *converges for the usual metric on* $\mathbb{R}^p$, i.e. $\lim_{n\to\infty} \Lambda_A(X_n) = S$, and $X \sqsubseteq_A \Upsilon_A(S)$.

Intuitively, these functions transpose the convergence of the sequence of iterates into the theory of real sequences, in such a way that the real sequence does not lose any information. Note that the order on $\mathbb{R}^p$ induced by the usual metric is unrelated with the order $\sqsubseteq_A$ on A, so the notion of extraction and combination is different from the notion of Galois connection used to compare abstract domains. Dire que les deux fonctions d'extraction et de combinaison sont fortement lies aux domaines abstraits utilisés, ils doivent être définit séparèment pour chaque domaine et ainsi ils doivent vérifier les deux propriétés afin de garantir la non perte d'information.

Proposition1
Proposition2
Les expliqués prièvement.
Donner les deux exemples pour les domaines des intervalles et octogones.
1- Domaine des intervalles: Définition + Les fonctions extraction et combinaison + les preuves.
2- Domaine des octogones: Définition + Les fonctions extraction et combinaison + les preuves

For the interval domain $I = \mathbb{I}^v$, where v is the number of variables of the program and $\mathbb{I}$ is the set of floating-point intervals, the extraction and the combination functions are defined in Equation (5).

$$\Lambda_I : \begin{cases} I \to \mathbb{R}^{2v} \\ (i_1, \ldots, i_v) \mapsto (\overline{i_1}, \underline{i_1}, \ldots, \overline{i_v}, \underline{i_v}) \end{cases} \qquad \Upsilon_I : \begin{cases} \mathbb{R}^{2v} \to I \\ (x_1, x_2, \ldots, x_{2v-1}, x_{2v}) \mapsto ([x_1, x_2], \ldots, [x_{2v-1}, x_{2v}]) \end{cases} \tag{5}$$

For other domains, these functions must be designed specifically. For example, we believe that such functions can be easily defined for the octagon abstract domain [**?**]: the function Λ associates a vector containing all its coefficients with a *difference bound matrix*. Special care should be taken in the case of infinite coefficients. More generally, we believe that for domains with a predefined shape, the functions Λ and Υ can be easily defined. Note that if there is a Galois connection (α_I, γ_I) between a domain A and the interval domain I, the extraction and combination functions can be defined as $\Lambda_A = \Lambda_I \circ \alpha_I$ and $\Upsilon_A = \gamma_I \circ \Upsilon_I$. We use this method in the last experiment in Section 5.2.

4.2 Accelerated abstract fixpoint computation

We describe the insertion of acceleration methods in the Kleene iteration process in Algorithm 1. We compute in parallel the sequence (X_n) coming from the Kleene's iteration and the accelerated sequence $(\boldsymbol{y}_n)$ computed from an accelerated method. Once the sequence $(\boldsymbol{y}_n)$ seems to converge, that is, the distance between two consecutive elements of $(\boldsymbol{y}_n)$ is smaller than a given value δ, we combine the two sequences. That is we compute the upper bound of the two elements of the current iteration. Note that the monotonicity of the computed sequence (X_n) is still guaranteed. Expliquer avec plus de détails l'algorithme et dire qu'on ne change pas l'algo initial mais on ajoute des améliorations.

The use of acceleration methods may be seen as an automatic delayed application of the widening with thresholds. Let us remark that we are not guaranteed to terminate in finitely many iterations: we know that asymptotically, the sequence $\boldsymbol{y}_i$ from Algorithm 1 gets closer and closer to the fixpoint,

Algorithm 1 Accelerated abstract fixpoint computation

1: **repeat**
2: $X_i := X_{i-1} \sqcup F(X_{i-1})$
3: $\boldsymbol{y}_i := \text{Accelerate}\,(\Lambda_A(X_0), \ldots, \Lambda_A(X_i))$
4: **if** $||\boldsymbol{y}_i - \boldsymbol{y}_{i-1}|| \leq \delta$ **then**
5: $X_i := X_i \sqcup \Upsilon_A(\boldsymbol{y}_i)$
6: **end if**
7: **until** $X_i \sqsubseteq X_{i-1}$

but we are not guaranteed that it reaches it. To guarantee termination of the fixpoint computation, we have to use more "radical" widening thresholds, for example after n applications of the accelerated method. So this method cannot be a substitute for widening, but it improves it by reducing the number of parameters (delay and thresholds) that a user must define.

5 Experimentation

To illustrate our acceleration methods, we used a simple static analyzer [3] working on the interval abstract domain that handles C programs without pointers. Furthermore, we associated to the analyzer our OCaml library of acceleration methods that transform an input sequence (given as a sequence of values) into its accelerated version. The obtained results are presented in the following sections. Note that we tested all these examples on the Interproc analyzer [4] . In each case, with standard parameters, the widening removed any constraints on the interesting variables. We only could obtain precise results by replacing the widening by a join, i.e. computing the standard Kleene iteration without widening.

5.1 Butterworth order 1

To test the acceleration method, we use a first-order Butterworth filter (see Figure 5, left). This filter is designed to have a frequency response which is as flat as mathematically possible in the band-pass and is often used in embedded systems to treat the input signals for a better stability of the program.

The static analysis of this program using the interval abstract domain defines 10 sequences, two for each variable (`x1`, `xn1`, `y`, `u`, `i`). These sequences converge toward the smallest fixpoint after a lot of iterations, our acceleration methods allow us to obtain the same fixpoint faster. In this example, we accelerate just the upper bound sequences because the lower ones are constant for

[3] This analyzer is based on Newspeak, http://penjili.org/newspeak.html.
[4] http://pop-art.inrialpes.fr/interproc/interprocweb.cgi. The SIMPLE programs corresponding to these examples can be found at www.lix.polytechnique.fr/~bouissou/NSAD10.

```
x1 = 0; y = 0; xn1 = 1;
for (i=0;i<200;i++) {
  /*!npk u between 1 and 2 */
  xn1 = 0.90480*x1 + 0.95240*u;
  y = 0.09524*x1 + 0.04762*u;
  x1 = xn1;
}
```

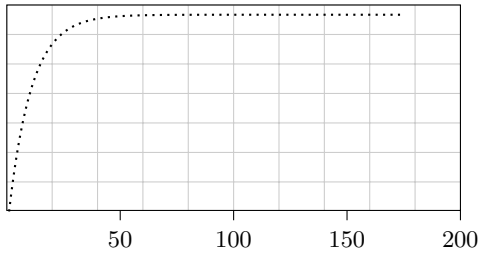


Fig. 5. The Butterworth program (left) and the sequence of supremum of variable `x1` (right).

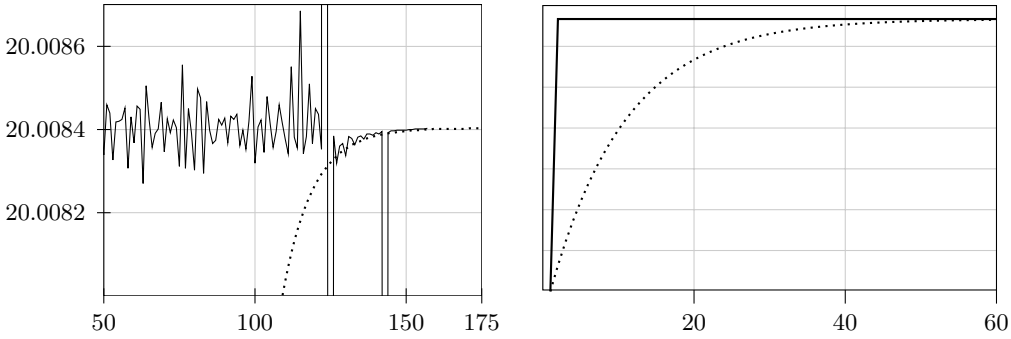


Fig. 6. Accelerated sequences (in bold) compared with the original Kleene sequence (dotted). Left is the sequence obtained with Aitken (zooming on the numerical problems), right with the ε-algorithm (zooming on the first iterates).

all the variables. We next present the result obtained with different methods on the variable `x1` only, results obtained with other variables are very alike.

The Aitken Δ^2-method. In Figure 5, right, with Kleene iteration and without widening, this program converges in 156 iterations, and we get the invariant $[0, 20.0084]$ for `x1`. With the Aitken Δ^2-method, we obtain only in 3 iterations a value very close to 20.0084, but problems of numerical instabilities prevent the stabilization of the program. However the values of the accelerated sequence stay in the interval $[20.0082, 20.0086]$ between the third and the last iteration (see Figure 6, left), which is a good estimate of the convergent point.

The ε-algorithm. In Figure 6, right, we notice a important improvement in the computation of the fixpoint, thanks to the ε-algorithm. With this method, the fixpoint of the variable `x1` is approximated with a precision of 10^{-6} after exactly 8 iterations, while Kleene iteration needed 156 steps. Note that to obtain 8 elements of the accelerated sequence we need 15 elements from the initial one. We obtain the same results with the vector ε-algorithm.

5.2 Butterworth order 2

An order 2 Butterworth filter is given by the following recurrence equation, where $\boldsymbol{x}_n$ is a two-dimensional vector, $\boldsymbol{x}_n = (x_1, x_2)^T$:

$$\boldsymbol{x}_{n+1} = \begin{pmatrix} 0.9858 & -0.009929 \\ 0.00929 & 1 \end{pmatrix} \cdot \boldsymbol{x}_n + u \cdot \begin{pmatrix} 0.9929 \\ 0.004965 \end{pmatrix}, \quad y_{n+1} = \begin{pmatrix} 4.965e^{-5} \\ 0.01 \end{pmatrix} \cdot \boldsymbol{x_n} + 2.482e^{-5} \cdot u$$

On this program, the results obtained using the interval abstract domain are not stable. To address this problem we have used Fluctuat [**?**], a static analyzer using a specific abstract domain based on affine arithmetic, a more accurate extension of interval arithmetic. It returns the upper and lower bounds of each variable. We applied the vector ε-algorithm on this example with 3 different values of δ (see Algorithm 1): this gives Figure 7. For example, for the variable x_1 and $\delta = 10^{-3}$, the over-approximation of the fixpoint is reached after 26 iterations (6 iterations before re-injection and 20 iterations after). Note that we obtain the same fixpoint as with Kleene iteration. We notice that the performance of the Algorithm 1 does not strongly depend on δ. Until now, we use the acceleration just once (unlike in Algorithm 1), a full implementation of it will probably reduce the number of iterations even more.

6 Related work

Most of the work in abstract interpretation based static analysis concerned the definition of new abstract domains (or improvements of existing ones), and the abstract fixpoint computation remained less studied. Initial work from Cousot and Cousot [**?**] discussed various methods to define widening operators. Bourdoncle [**?**] presented different iteration strategies that helps to reduce the over-approximation introduced by widening. These methods are complementary to our technique: as explained in Section 4, acceleration should be done at the same control point as the one chosen for widening, and does not replace standard widening as the termination of the fixpoint computation is not guaranteed. However, acceleration methods greatly improve widening by dynamically and automatically finding good thresholds.

Gopan and Reps in their *guided static analysis* framework [**?**,**?**] also used the idea of computing in parallel the main iterates and a guide that shows where the iterates are going. In their work, the precision of the fixpoint computation is increased by computing a *pilot value* that explores the state

Variable	Kleene	Vector ε-algorithm (`Before` + `After`)		
		$\delta = 10^{-3}$	$\delta = 10^{-4}$	$\delta = 10^{-5}$
x_1	70	7 (6 + 1)	9 (8 + 1)	22 (16 + 6)
x_2	83	26 (6 + 20)	23 (8 + 15)	17 (16 + 1)
y	83	26 (6 + 20)	23 (8 + 15)	19 (16 + 3)

`Before`: number of iterations to reach the condition on δ. `After`: the remaining number of Kleene iterations to reach the invariant using the accelerated result.

Fig. 7. Numbers of iterations needed to reach an invariant.

space using a restricted version of the iteration function. Once this pilot has stabilized, it is used to accelerate the main iterates; in a sense, this pilot value is very similar to the value $\boldsymbol{y}_i$ of Algorithm 1, but we do not modify the iteration function as done in [**?**].

Maybe the work that is the closest to ours is the use of acceleration techniques in model checking [**?**], that have recently been applied to abstract interpretation [**?**,**?**]. In this framework, the term acceleration is used to describe techniques that try to predict the effect of a loop on an abstract state: the whole loop is then replaced with just one transition that safely and precisely approximates it. These techniques perform very well for sufficiently simple loops working on integer variables, and gives exact results for such cases. Again, this method is complementary to our usage of acceleration: it *statically* modifies the iteration function by replacing simple loops with just one transition, while our method *dynamically* predicts the limit of the iterates. We believe that our method is more general, as it can be applied to many kinds of loops and is not restricted to a specific abstract domain (changing the abstract domain only requires changing the Λ_A and Υ_A functions).

Note also that the computation of symbolic loop invariants such that [**?**] produce precise results. Nevertheless, they have to limit the constructions of analyzed programs unlike our approach.

7 Conclusion

We presented a technique to accelerate abstract fixpoint computations using numerical acceleration methods. This technique consists in building numerical sequences by extracting, at every iteration, supremum and infimum from every variable of the program. To the obtained sequences we apply the various convergence acceleration methods, which allows us to get significantly closer or to reach the fixpoint more quickly than the Kleene iteration. To make sure that the fixpoint returned by the accelerated method is indeed the fixpoint of the abstract semantics, we re-inject it in the static analyzer. This guarantees us the fast stop of the analyzer with a good over-approximation of the fixpoint. The experiments made on a certain number of examples (linear programs) show a good acceleration of the fixpoint computation especially when we use the ε-algorithm, where the number of iterations is divided by four. Let us note that we have assumed in this article that the sequences of iterates and the corresponding vector sequences converge towards a finite limit. In case of diverging sequences, traditional widening can be used as sequence transformation will not perform as well as for converging ones.

For now, we made the experimentation using two separate programs: one that computes the Kleene iterates, and one that accelerates the sequences. The Algorithm 1 is thus still not fully implemented, its automatization is

the object of our current work. Since the use the interval abstract domain allows us to cover just a small set of programs, our future work will also consist in extending this technique to relational domains such as octagons and polyhedra. Moreover, we presented examples made of a simple loop that iterates a linear transformation, we will test our techniques on more realistic programs. Early experiments show that acceleration behaves well on loops iterating a non-linear function. It will be more difficult to treat loops that are less regular, e.g. loops with if statements, as the extracted sequences are less regular. However we believe that our technique can be mixed with guided static analysis [**?**] to achieve a precise and efficient result in these cases.

Acknowledgement

The authors want to thank X. Allamigeon and E. Goubault for their helpful discussions and precious advices and S. Zennou for her technical help on the analyzer based on Newspeak. The authors are also thankful to the anonymous reviewers for their helpful comments and suggestions.