Static Analysis of Finite Precision Computations

Eric Goubault and Sylvie Putot
CEA LIST, Laboratory for the Modelling and Analysis of Interacting Systems,
Point courrier 94, Gif-sur-Yvette, F-91191 France, Firstname.Lastname@cea.fr

Abstract. We define several abstract semantics for the static analysis of finite precision computations, that bound not only the ranges of values taken by numerical variables of a program, but also the difference with the result of the same sequence of operations in an idealized real number semantics. These domains point out with more or less detail (control point, block, function for instance) sources of numerical errors in the program and the way they were propagated by further computations, thus allowing to evaluate not only the rounding error, but also sensitivity to inputs or parameters of the program. We describe two classes of abstractions, a non relational one based on intervals, and a weakly relational one based on parametrized zonotopic abstract domains called affine sets, especially well suited for sensitivity analysis and test generation. These abstract domains are implemented in the Fluctuat static analyzer, and we finally present some experiments.

1 Introduction

In this article, we discuss several abstract domains for proving properties about the potential loss of accuracy in numerical programs using finite-precision arithmetics, such as IEEE 754 floating-point numbers, fixed point semantics, and even integers with finite range. The goal of such abstractions is not to find runtime errors, but rather to automatically prove the the computation made by a program (using finite-precision arithmetics) conforms to what was expected by the programmer (using real-number semantics). These properties are of utmost importance in the field of embedded systems (see the Patriot bug for instance [27]) and in computer architecture and numerical simulation (see for instance [24]). Take as explanatory example the following simple C program:

```
1   float x=[0,1]; float y=(x-1)*(x-1)*(x-1)*(x-1);
2   float z=x*x;
3   float z=z*z-4*x*z+6*z-4*x+1; float t=z-y;
```

Using our analysis, we will find automatically (in our fully-fledged analyzer Fluctuat, in less than 0.01 seconds) that y is in the real number semantics within 0 and 1 with a negligible error in the floating-point number semantics in $[-4.2.10^{-7}, 4.2.10^{-7}]$ whereas z is in $[-1.70, 2.25]$ (real number semantics) with an error in the floating-point number semantics of $[-2.1.10^{-6}, 2.1.10^{-6}]$ mostly due to line 3. We will actually see in Section 5 that using other mechanisms, implemented in Fluctuat [5], we can improve a lot these bounds and even prove that t is almost 0, showing that z is the same calculation in real numbers as y. On other examples, we will also demonstrate the use of Fluctuat to hint at less well behaved rounding error.

This paper is bridging the gaps between the initial proposal [11] and the actual implementation in the Fluctuat tool as demonstrated in [17, 3, 5]. It describes in particular the relational abstraction of the imprecision errors, never published before.

Related Work This article is linked with earlier work by the authors, in particular concerning the relational abstractions of real numbers using zonotopes [15, 16] and is improving on [11, 14].

Other proposals have been made to analyze imprecision errors, most notably [1], in which floating-point variables are represented as a triplet: its floating-point value, its value computed using the real-number semantics, and its intended value that the programmer ultimately wanted to compute. The first two components correspond to our semantic model, although the error is not decomposed along the history of computation in [1]. The last component allows for computing the “model error” and not only the “implementation error”, but this is also feasible with our relational abstraction described in Sections 4.1 and 4.2, see for instance Example 3. Finally, our approach uses abstract interpretation and thus is fully automatic, whereas the approach of [1] is based on Hoare proofs and thus is interactive.

More generally, the subject of analyzing the floating-point number semantics has gained importance in the early 2000s in fully-fledged static analyzers. A common approach to go from a real number abstraction to a floating-number abstract semantics is to use linearization, see [20, 21], as implemented in APRON [25]. This approach allows for correctly abstracting the floating-point semantics, but does not decompose it into its real number and error decomposition. Also, the approach we are taking in Section 4.2 allows for finer results, even though the computation of the decomposition of errors along the computation history makes results less precise when we ask for finer information of the location of these errors. For instance, we only find with these techniques, using linearizations of polyhedra (APRON/Polka): y in $[0, 1]$ in real numbers, in $[-5.96.10^{-7}, 1.00000059604653]$ in the floating-point semantics, but no relation between the two abstract semantic values, and z in $[-7, 6.75]$ in real numbers, and in $[-3.000002384, 5.7500668]$ in floating-point numbers and no relation between real and floating-point values. So we are even unable to prove with linearizations of polyhedra that the imprecision error, notwithstanding its origin, is small. Even the concretization of y obtained by our technique is better (in $[-4.17.10^{-7}, 1.000000417]$) and much better for z indeed.

Contents We begin by describing briefly the concrete semantics of finite-precision computations, focusing mostly on floating-point numbers, in Section 2. This semantics is non-standard in that it attributes to each variable x a triplet (f^x, r^x, e^x) where f^x (resp. r^x, e^x) is its floating-point (resp. real number, error) value. In Section 3, we detail the first natural abstraction, by intervals, of the values and the decomposition of error terms. We then describe in Section 4 a more accurate zonotopic abstraction, using affine sets, of the values and errors.

We will see that different choices of abstraction lead to a different way to see the triplet (f^x, r^x, e^x). With intervals, f^x will be first computed, then we will deduce e^x. Finally, r^x can be deduced by $e^x + f^x$, or computed directly. Whereas relational abstractions naturally apply to real numbers, but not to their finite-precision approximations: we will first compute r^x, then deduce e^x and f^x. We discuss in Section 5 the implementation of these abstractions, and show detailed examples and benchmarks.

2 Modelling finite precision computations

We insist here more specifically on the modelling of more standard floating-point numbers; however the domains presented hereafter are perfectly well suited to the handling of fixed-point numbers. Indeed, an abstract domain for the analysis of programs in fixed-point numbers is implemented in our static analyzer Fluctuat.

2.1 Finite precision computations

Floating-point arithmetic We recall here very briefly some basic properties of floating-point arithmetic, and refer the reader to [10, 22] for instance for more details. The IEEE 754 standard [7] specifies the way most processors handle finite-precision approximation of real numbers known as floating-point numbers. It specifies four rounding modes. When the rounding mode is fixed, it then uniquely specifies the result of rounding of a real number to a floating-point number. Let $\uparrow\colon \mathbb{R} \rightarrow \mathbb{F}$ be the function that returns the rounded value of a real number r. Whatever the rounding mode and the precision of the floating-point analyzed are, there exist two positive constants δ_r and δ_a (which value depend on the format of the floating-point numbers), such that the error $r^x - \uparrow r^x$ when rounding a real number r^x to its floating-point representation $\uparrow r^x$ can be bounded by

$$|r^x - \uparrow r^x| \leq max(\delta_r |\uparrow r^x|, \delta_a). \tag{1}$$

For any arithmetic operator on real numbers $\diamond \in \{+, -, \times, /\}$, we note $\diamond_{\mathbb{F}}$ the corresponding operator on floating-point numbers, with rounding mode the one of the execution of the analyzed program. IEEE-754 standardises these four operations, as well as the square root: the result of the floating-point operation is the same as if the operation were performed on the real numbers with the given inputs, then rounded.

We are not interested in run-time errors or exceptions: when an overflow or undefined behaviour is encountered, our analysis simply reports $\top$ as a result.

Integers Machine integers have finite range: we consider modular arithmetic, where adding one to the maximum value representable in a given data type gives the minimum value of this data type. The difference with natural integers is an error term in our model (the "real" value being the natural integer).

Fixed-point arithmetic Fixed-point numbers are a finite approximation of real numbers with a fixed number of digits before and after the radix. They are essentially integer numbers scaled by a specific factor determined by the type. The main differences with floating-point arithmetic are that the range of values is very limited, and that the absolute rounding error is bounded and not the relative error. As the only properties we will use in our abstractions, are a function giving the rounded value $\uparrow r^x$ of a real number r^x, and a range for the rounding error $r^x - \uparrow r^x$ (no abstraction of the relative error is used), all the work below naturally applies to the analysis of fixed-point arithmetic.

2.2 Concrete model

The underlying idea of the concrete model, first sketched in [11], then further described in more details in [18], and meanwhile implemented in a first version of the Fluctuat static analyzer [12], is to describe the difference of behaviour between the execution of a program in real numbers and in floating-point numbers. For that, the concrete value of a program variable is a triplet (f^x, r^x, e^x), where $f^x \in \mathbb{F}$ is the value of the variable if the program is executed with a finite-precision semantics, r^x is the value of the same variable if the program is executed with a real numbers semantics, and e^x is the rounding error, that is $e^x = r^x - f^x$. A variation of the same idea is used in the context of formal proof in [1], where the concrete model of a floating-point variable relies on its floating-point and idealized real values.

Another idea of [11, 18, 12], also developed here, is that it could be of interest for a static analysis to decompose the error term e^x along its provenance in the source code of the analyzed program, in order to point out the main sources of numerical discrepancy. For that, depending on the level of detail required, control points, blocks, or functions of a program can be annotated by a label ℓ, which will be used to identify the errors introduced during a computation. We note $\mathcal{L}$ the set of labels of the program, and $\mathcal{L}^+$ the set of words over these labels, then we express the error term as a sum

$$e^x = \bigoplus_{u \in \mathcal{L}^+} e_u, \tag{2}$$

where $e_u \in \mathbb{R}$ is the contribution to the global error of operations involved in word u. A word of length n thus identifies an order n error, which originates from several points in the program, and as for labels, we can choose different levels of abstraction of multiple sources of errors. We refer to [18] for details about this.

Our concrete model also models integers. The semantics of machine integers does not introduce rounding errors, but there are two sources of error terms. First, the effect of rounding errors on a floating-point variable that is cast in an integer i, is considered as an error term e^i. Second, the effect of the finite representation of integers by modular arithmetic is also expressed as an error term, as compared to a computation with infinite integers.

We will now describe two different abstractions of this concrete model.

3 A non relational abstraction

3.1 Interval abstraction of values and error decomposition

The first natural idea to get an efficient abstract domain of our concrete model, is to abstract the values and errors with intervals, and to agglomerate all errors of order greater than one in a remaining interval term. This is what was first implemented in the Fluctuat static analyzer, and partially described in [12].

Correctness of the transfer functions is based on the separate Galois connection abstraction on each of the component of the triplet (f^x, r^x, e^x) of the collecting semantics based on the concrete semantics of Section 2.2.

Interval arithmetic We denote intervals by bold letters. For two intervals with real bounds $\boldsymbol{x} = [\underline{x}, \overline{x}]$ and $\boldsymbol{y} = [\underline{y}, \overline{y}]$, we define:

$$\boldsymbol{x} + \boldsymbol{y} = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]; \quad \boldsymbol{x} - \boldsymbol{y} = [\underline{x} - \overline{y}, \overline{x} - \underline{y}]$$

$$\boldsymbol{x} \times \boldsymbol{y} = [\min(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}), \max(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y})]$$

The same operators can also be defined over finite precision approximations of the real numbers (typically floating-point numbers), for the current rounding mode. Finally, for the implementation, we need to define an abstraction of interval arithmetic over real numbers: we thus need the same operators defined over the set of floating-point of arbitrary precision p, noted $\mathbb{F}_p$, with outward rounding:

$$\boldsymbol{x} +_{\mathbb{F}_p} \boldsymbol{y} = [\underline{x} +_{\mathbb{F}_p}^{-\infty} \underline{y}, \overline{x} +_{\mathbb{F}_p}^{\infty} \overline{y}], \tag{3}$$

where $+_{\mathbb{F}_p}^{-\infty}$ (resp. $+_{\mathbb{F}_p}^{\infty}$) denotes the plus operator on floating-point numbers with an arbitrary precision p and rounding towards minus infinity (resp. rounding towards plus infinity).

Abstract model An abstract element x is a triplet

$$x = \left(\boldsymbol{f}^x, \ \boldsymbol{r}^x, \ \bigoplus_{l \in \mathcal{L}} \boldsymbol{e}_l^x \oplus \boldsymbol{e}_{ho}^x \right) \tag{4}$$

where $\boldsymbol{f}^x = [\underline{f^x}, \overline{f^x}]$ bounds the finite precision value, with $(\underline{f^x}, \overline{f^x}) \in \mathbb{F} \times \mathbb{F}$, $\boldsymbol{r}^x = [\underline{r^x}, \overline{r^x}]$ bounds the real value, with $(\underline{r^x}, \overline{r^x}) \in \mathbb{R} \times \mathbb{R}$, $\boldsymbol{e}_l^x = [\underline{e_l^x}, \overline{e_l^x}]$ bounds the first-order contribution of control point l on the error between the real and the computed value of variable x, and $\boldsymbol{e}_{ho}^x = [\underline{e_{ho}^x}, \overline{e_{ho}^x}]$ the sum of all higher-order contributions (most of the time negligible), with $(\underline{e_l^x}, \overline{e_l^x}) \in \mathbb{R} \times \mathbb{R}$, for all l in $\mathcal{L} \cup ho$. The first-order errors are the propagated elementary rounding errors that can be associated to a specific control point. Higher-order errors appear in non affine arithmetic operations, and are non longer associated to a specific control point: they occur for instance when multiplying two error terms. The sum of the error intervals over-approximates the global error e^x due to finite precision computation or to initial errors on inputs.

3.2 Transfer functions for arithmetic operations

Constants and inputs For an interval $\boldsymbol{r}^x = [\underline{r^x}, \overline{r^x}]$, we note $\uparrow \boldsymbol{r}^x = [\uparrow \underline{r^x}, \uparrow \overline{r^x}]$. Using bound (1), we can over-approximate the rounding error of a real value given in interval $\boldsymbol{r}^x$ to its finite precision representation, by the interval

$$\boldsymbol{e}(\boldsymbol{r}^x) = [-u^x, u^x] \cap (\boldsymbol{r}^x - \uparrow \boldsymbol{r}^x), \tag{5}$$

where $u^x = \max(\delta_r \max(|\uparrow \underline{r^x}|, |\uparrow \overline{r^x}|), \delta_a)$. This expresses the fact that we can compute the error as the intersection of the abstraction of bound given by (1), and the actual difference between the real and the finite precision values. The right-hand side or the left-hand side will be more accurate in different cases, depending for instance on the width of the range of values abstracted.

Arithmetic operations Then, using this abstraction of the rounding error with the fact that the arithmetic operations on floating-point operations are correctly rounded (Section 2.1), we define the transfer function for expression $z = x \diamond^n y$ at label n. When $\diamond$ is the plus or minus operator, we have:

$$z = (\boldsymbol{f}^x \diamond_{\mathbb{F}} \boldsymbol{f}^y,\ \boldsymbol{r}^x \diamond \boldsymbol{r}^y,\ \bigoplus_{l\in\mathcal{L}\cup ho} (\boldsymbol{e}_l^x \diamond \boldsymbol{e}_l^y) \oplus \boldsymbol{e}(\boldsymbol{f}^x \diamond \boldsymbol{f}^y)),$$

where the new error term $\boldsymbol{e}(\boldsymbol{f}^x \diamond \boldsymbol{f}^y)$ is associated to label n. Indeed, the error on the result of an arithmetic operation combines the propagation of existing errors on the operands, plus a new round-off error term.
For the multiplication, we define:

$$z = (\boldsymbol{f}^x \times_{\mathbb{F}} \boldsymbol{f}^y,\ \boldsymbol{r}^x \times \boldsymbol{r}^y,\ \bigoplus_{l\in\mathcal{L}\cup ho} (\boldsymbol{f}^x \times \boldsymbol{e}_l^y + \boldsymbol{f}^y \times \boldsymbol{e}_l^x) \oplus \sum_{(l,k)\in(\mathcal{L}\cup ho)^2} \boldsymbol{e}_l^x \boldsymbol{e}_k^y \oplus \boldsymbol{e}(\boldsymbol{f}^x \times \boldsymbol{f}^y)).$$

The semantics for the division x/y is defined using a first-order Taylor expansion to compute the inverse of $f^y \bigoplus_{l\in\mathcal{L}\cup ho} \boldsymbol{e}_l^y$: the approximation error is bounded and seen as an additional error. This expansion is all the more accurate as the errors are small compared to the value.

Integer computations The cast of a floating-point (or fixed-point) number to an integer, as long as it does not result in an overflow of the integer result, is not seen as an additional error, so that the real value r^x is also cast in an integer. Let us note $(int)(r^x)$ the result of casting a real variable to an integer, and $(int)(\boldsymbol{a})$ its extension to an interval $\boldsymbol{a}$, then we define $i = (int)x$ by:

$$i = \left((int)(\boldsymbol{f}^x),\ (int)(\boldsymbol{r}^x), \left(\sum_{l\in\mathcal{L}\cup ho} \boldsymbol{e}_l^x + [-1,1]\right) \cap ((int)(\boldsymbol{r}^x) - (int)(\boldsymbol{f}^x))\right),$$

where the error is fully assigned to the label of the rounding operation: the sources of errors in previous computations are thus lost, we only keep track of the fact that rounding errors on floating-point computation can have an impact on an integer variable.

The addition, subtraction and multiplication on integer variables can be directly extended from their floating-point version (same propagation of existing errors, no additional rounding error, but an additional error of MAX_INT-MIN_INT if an overflow occurs). The integer division can be interpreted by a division over reals (error propagation with no additional error) followed by a cast to an integer.

3.3 Order-theoretic operations

Join The most natural join operation between two abstract values is to define it by component-wise join on the intervals:

$$x \cup y = \left(\boldsymbol{f}^x \cup \boldsymbol{f}^y,\ \boldsymbol{r}^x \cup r^y,\ \bigoplus_{l\in\mathcal{L}} (\boldsymbol{e}_l^x \cup \boldsymbol{e}_l^y) \oplus (\boldsymbol{e}_{ho}^x \cup \boldsymbol{e}_{ho}^x)\right),$$

where the join of two intervals $\boldsymbol{a}$ and $\boldsymbol{b}$ is $\boldsymbol{a}\cup\boldsymbol{b} = [\min(\underline{a},\underline{b}), \max(\overline{a},\overline{b})]$. This join operation presents an inconvenience: when joining abstract values coming from different control flows, the joined abstract value will contain the elementary errors coming from these different control flows, so that the sum of the elementary errors of this joined value can be larger than when first computing the sums of errors then joining the result. In practice, in order to overcome this problem, we keep in the abstract domain an interval representing the global error, that is obtained by interval join of the global error, and which is thus more accurate than the decomposed sum.

Test interpretation Our analysis relies on the assumption that the control flow of the program is the same for the finite precision and real values of the program. If this is found not to be the case when evaluating a boolean condition in real and in floats, an unstable test is indicated. The finite precision control flow is followed: this can give unsound error bounds, but the user is indicated which test to look at. A solution to avoid this assumption is to follow the two control flows and at the next join, merge the abstract values, by adding to the abstract value of the finite precision control-flow, the difference with the value of the real value control-flow as a rounding error. However, this can become quite costly and will not be expanded further in this paper.

Now, with the assumption that the finite precision and real values take the same control-flow, we define the meet operation by $x \cap y = (\boldsymbol{f}^x \cap \boldsymbol{f}^y,\ \boldsymbol{r}^x \cap \boldsymbol{r}^y,\ \boldsymbol{e}^x)$, where the meet of two intervals $\boldsymbol{a}$ and $\boldsymbol{b}$ is $\boldsymbol{a} \cap \boldsymbol{b} = [\max(\underline{a}, \underline{b}), \min(\overline{a}, \overline{b})]$. (by convention, the interval which lower bound is greater than the upper bound is the empty interval) The hypothesis that the finite precision and real control flow are the same also allows to reduce the error terms.

3.4 Fixpoint iteration

In order to ensure termination of the fixpoint computation, we define the following natural widening operator on abstract values, by applying the componentwise classical interval widening on each elementary terms:

$$x \nabla y = \left(\boldsymbol{f}^x \nabla \boldsymbol{f}^y,\ \boldsymbol{r}^x \nabla r^y,\ \bigoplus_{l \in \mathcal{L}} (\boldsymbol{e}^x_l \nabla \boldsymbol{e}^y_l) \oplus (\boldsymbol{e}^x_{ho} \nabla \boldsymbol{e}^x_{ho}) \right),$$

where the widening on intervals is for instance the classical one proposed of [4]. Note that it is most of the time necessary to operate a semantic loop unrolling to ensure a good convergence of fixpoint computations [16].

4 A zonotopic abstraction

The abstraction of Section 3 naturally suffers from the over-estimation of interval analysis. We will now present an abstraction of the triplet (f^x, r^x, e^x) relying on a zonotopic weakly-relational abstract domain for the analysis of real value variables, based on ideas from affine arithmetic [2]: these domains that we developed over the last few years use a parametrization of zonotopes we call affine sets, that allow an accurate and computationally efficient functional abstraction of input-output relations on the values of real variables [14, 15, 8, 9].

As the good algebraic properties of real numbers do not hold on floating-point (or fixed-point) number, the relational domains do not apply directly on finite precision values. We will present here how we can use them to bound the real value r^x and the error e^x of the triplet (f^x, r^x, e^x). From this, the finite precision value of variables can be bounded by the reduced product of $r^x - e^x$ with the intersection of a direct interval computation of f^x. Hints on previous work on the subject were given in [26, 13, 14], but never actually described a relational abstraction of the error terms.

After a quick introduction in Section 4.1 to affine sets for real value estimation, we focus in section 4.2 on their use to abstract the full value (f^x, r^x, e^x).

4.1 Zonotopic abstract domain for real values

Affine arithmetic [2] is a more accurate extension of interval arithmetic, that takes into account linear correlation between variables. An affine form $\hat{x}$ is a formal sum over a set of noise symbols ε_i: $\hat{x} = \alpha_0^x + \sum_{i=1}^n \alpha_i^x \varepsilon_i$, where $\alpha_i^x \in \mathbb{R}$ and the noise symbols ε_i are independent symbolic variables with unknown value in $[-1,1]$. The coefficients $\alpha_i^x \in \mathbb{R}$ are the partial deviations to the center $\alpha_0^x \in \mathbb{R}$ of the affine form. These deviations can express uncertainties on the values of variables, for instance when inputs or parameters are given in a range of values, but also uncertainty coming from computation. The sharing of the same noise symbols between variables expresses implicit dependency. The values that a variable x defined by an affine form $\hat{x}$ can take is in the range

$$\gamma(\hat{x}) = \left[\alpha_0^x - \sum_{i=1}^n |\alpha_i^x|, \alpha_0^x + \sum_{i=1}^n |\alpha_i^x|\right]. \tag{6}$$

Assignment The assignment of a variable x whose value is given in a range $[a,b]$, is defined as a centerd form using a fresh noise symbol $\varepsilon_{n+1} \in [-1,1]$, which indicates unknown dependency to other variables: $\hat{x} = \frac{(a+b)}{2} + \frac{(b-a)}{2}\varepsilon_{n+1}$.
Affine operations The result of linear operations on affine forms is an affine form. For two affine forms $\hat{x}$ and $\hat{y}$, and a real number λ, we get

$$\lambda\hat{x} + \hat{y} = (\lambda\alpha_0^x + \alpha_0^y) + \sum_{i=1}^n (\lambda\alpha_i^x + \alpha_i^y)\varepsilon_i$$

Multiplication For non affine operations, we select an approximate linear resulting form, and bounds for the error committed using this approximate form are computed, that create a new noise term added to the linear form:

$$\hat{x}\hat{y} = \alpha_0^x\alpha_0^y + \sum_{i=1}^n (\alpha_i^x\alpha_0^y + \alpha_i^y\alpha_0^x)\,\varepsilon_i + \left(\sum_{i=1}^n |\alpha_i^x\alpha_i^y| + \sum_{i<j}^n |\alpha_i^x\alpha_j^y + \alpha_j^x\alpha_i^y|\right)\varepsilon_{n+1}.$$

The joint concretization of these affine forms is a center-symmetric polytope, that is a zonotope. We defined in [15, 8, 9] abstract domains based on extensions of these affine forms, with an order relation, and corresponding join and meet operators. In particular, we defined in [9] a meet operation using a logical product of our affine sets with an abstract domain over the noise symbols: the constraints generated by the tests are interpreted over the noise symbols of the affine forms. We do not detail these operations here, we will refer in the rest of the paper to $\hat{x} \cup \hat{y}$ and $\hat{x} \cap \hat{y}$ for respectively the join and meet over two affine forms $\hat{x}$ and $\hat{y}$. The order relation on abstract values ensures the geometric ordering for the zonotope including the current variables and the inputs of the program, that is we have a functional abstraction of the behaviour of the program.

4.2 Abstract domain for finite precision computations

The triplet (f^x, r^x, e^x) is now abstracted using two sets of noise symbols: the ε_i^r that model the uncertainty on the real value, and the ε_i^e that model the uncertainty on the error. An abstract value consists of an interval and two affine forms: $x = (\boldsymbol{f}^x, \hat{r}^x, \hat{e}^x)$. The uncertainty on the real value also introduces an uncertainty on the error, which is partially modelled:

$$\hat{r}^x = r_0^x + \sum_i r_i^x \, \varepsilon_i^r$$

$$\hat{e}^x = e_0^x + \sum_i e_i^x \, \varepsilon_i^r + \sum_l e_l^x \, \varepsilon_l^e$$

In the error expression $\hat{e}^x$, $e_l^x \, \varepsilon_l^e$ expresses the uncertainty on the rounding error committed at point l of the program (its center being in e_0^x), and its propagation through further computations, while $e_i^x \, \varepsilon_i^r$ expresses the propagation of the uncertainty on value at point i, on the error term; it allows to model dependency between errors and values.

Transfer functions We now define the transfer function for expression $z = x \diamond^n y$ at label n. When $\diamond$ is the plus or minus operator, we define:

$$\begin{aligned}
\hat{r}^z &= \hat{r}^x \diamond \hat{r}^y, \\
\hat{e}^z &= \hat{e}^x \diamond \hat{e}^y + new_{\varepsilon^e}(\boldsymbol{e}(\gamma(\hat{r}^z - \hat{e}^x \diamond \hat{e}^y))), \\
\boldsymbol{f}^z &= (\boldsymbol{f}^x \diamond \boldsymbol{f}^y) \cap (\hat{r}^z - \hat{e}^z)
\end{aligned}$$

where γ is the interval concretization on affine forms defined by (6), interval function $\boldsymbol{e}$ is the interval abstraction of the rounding error defined in (5), and function new_{ε^e} creates a new error noise symbol: for an interval I, we define $new_{\varepsilon^e}(I) = mid(I) + dev(I)\varepsilon_{n+1}^e$, where ε_{n+1}^e is a fresh error noise symbol, $mid(\imath) = \frac{\underline{I}+\overline{I}}{2}$ denotes the center of the interval, and $dev(I) = \frac{\overline{I}-\underline{I}}{2}$ the deviation to its center. The affine form for the real value is obtained by the operation on affine forms $\hat{r}^x \diamond \hat{r}^y$ as defined in Section 4.1. Then the error is obtained by the sum of the propagation of the existing error $\hat{e}^x \diamond \hat{e}^y$ still by affine arithmetic, and of the new error of rounding the result in real numbers of $\hat{r}^z - \hat{e}^x \diamond \hat{e}^y$ to the floating-point result f^z.

For the multiplication, we define:

$$\begin{aligned}
\hat{r}^z &= \hat{r}^x \hat{r}^y, \\
\hat{e}^z &= \hat{r}^y \hat{e}^x + \hat{r}^x \hat{e}^y - \hat{e}^x \hat{e}^y + new_{\varepsilon^e}(\boldsymbol{e}(\gamma(\hat{r}^z - (\hat{r}^y \hat{e}^x + \hat{r}^x \hat{e}^y - \hat{e}^x \hat{e}^y)))), \\
\boldsymbol{f}^z &= (\boldsymbol{f}^x \boldsymbol{f}^y) \cap (\hat{r}^z - \hat{e}^z).
\end{aligned}$$

The real value is obtained by the multiplication on affine forms $\hat{r}^x \hat{r}^y$ as defined in Section 4.1. The propagation of the existing errors, obtained by expressing $\hat{r}^x \hat{r}^y - (\hat{r}^x - \hat{e}^x)(\hat{r}^y - \hat{e}^y)$, is computed by $\hat{r}^y \hat{e}^x + \hat{r}^x \hat{e}^y - \hat{e}^x \hat{e}^y$. The operations occurring in this expression are those over affine forms, with partially shared noise symbols between the real and error affine forms. When the propagation error is computed, we can finally deduce the new rounding error due to the multiplication by $new_{\varepsilon^e}(\boldsymbol{e}(\gamma(\hat{r}^z - (\hat{r}^y \hat{e}^x + \hat{r}^x \hat{e}^y - \hat{e}^x \hat{e}^y))))$.

Note that we chose not to separate the higher-order from the first-order term as in the non-relational semantics. Indeed, this decomposition would be more costly for no gain of accuracy. But most of all, we noticed that the higher-order terms tend to converge much more slowly in fixpoint iterations than if agglomerated as proposed here.

Using more refined properties of floating-point numbers Some specific properties of floating-point numbers can be used to refine the estimation of the new error in some cases. For instance, the well-known Sterbenz Theorem [28] that says that is x and y are two floating-point numbers such that $\frac{y}{2} \leq x \leq 2y$, then the result of $x - y$ is a floating-point number, so that no new rounding error is added in this case. This allows to refine the subtraction by writing $\hat{e}^z = \hat{e}^x - \hat{e}^x$ when the floating-point values of x and y satisfy the hypotheses of Sterbenz Theorem. Note that it is more accurate to verify the satisfaction of this condition on the affine forms $\hat{r}^x - \hat{e}^x$ and $\hat{r}^y - \hat{e}^y$ than on the intervals $\boldsymbol{f}^x$ and $\boldsymbol{f}^y$.

Example 1. Let x, y, and z be double precision floating-point numbers, and x initially given in range $[0, 2]$ for the following sequence of instructions:

```
x := [0,2];
y = 0.75*x;
z = x-y;
```

With the interval analysis, we get $\boldsymbol{f}^z = [-1.5, 2]$ and $\boldsymbol{e}^z = [-1.11e^{-16}, 1.11e^{-16}]_2 + [-2.22e^{-16}, 2.22e^{-16}]_3$. Indeed, as no relation is kept between x and y, the value of z is computed inaccurately, and so is the error, as it does not realize that Sterbenz theorem applies. Whereas with the relational analysis, $\hat{r}^x = 1 + \varepsilon_1^r$, $\hat{r}^y = 0.75 + 0.75\varepsilon_1^r$, $\hat{e}^y = 1.11e^{-16}\varepsilon_2^e$, and finally $\hat{r}^z = 0.25 + 0.25\varepsilon_1^r \in [0, 0.5]$ and $\hat{e}^z = -1.11e^{-16}\varepsilon_2^e \in [-1.11e^{-16}, 1.11e^{-16}]$ as Sterbenz theorem applies.

Casts from floating-point value to integers For lack of place, we only detail a very simple version of the cast. As already stated, the truncation due to the cast is not seen in itself as an error. So, the affine form for the real value is also cast to an integer, which results in a partial loss of relation, of amplitude bounded by 1. If the error on x was not zero, a loss of relation also applies. For the floating-point value, the cast directly applied to the interval value will be more accurate than $r^i - e^i$. The cast $i = (int)x$ then writes

$$\begin{aligned}
\hat{r}^i &= \hat{r}^x + new_{\varepsilon^r}([-1, 1]) \\
\hat{e}^i &= \begin{cases} 0 & \text{if } e^x = 0 \\ new_{\varepsilon^e}([-1, 1]) & \text{if } \gamma(e^x) \in [-1, 1] \\ e^x + new_{\varepsilon^e}([-1, 1]) & \text{otherwise} \end{cases} \\
\boldsymbol{f}^i &= (int)\boldsymbol{f}^x
\end{aligned}$$

Order-theoretic operations The join operation is computed component-wise, using the join over intervals and affine forms $x \cup y = (\boldsymbol{f}^x \cup \boldsymbol{f}^y, \hat{r}^x \cup \hat{r}^y, \hat{e}^x \cup \hat{e}^y)$. Note that we do not have here the over-estimation problem over errors that we had with the non-relational join. Indeed, the join we define over affine forms keeps only the greatest common relation (see [15, 8, 9]). It will thus lose the sources of errors that are not common to all execution paths of the programs, but will assign them to the label of the corresponding join operator.

As in the non relational abstraction, we make the assumption for the analysis that the control flow of the program is the same for the floating-point and real executions (see Section 3.3 for the discussion of this assumption), so that we interpret the tests over the real and floating-point values. For the meet operation, we thus interpret $\hat{r}^x \cap \hat{r}^y$ and $\hat{r}^x - \hat{e}^x \cap \hat{r}^y - \hat{e}^y$. Remember (it was briefly stated in Section 4.1, see [9] for more details) that the interpretation of tests yields constraints on the noise symbols. By lack of place, we will not detail this formally here.

Fixpoint computation, widening We described in [15, 8, 9] fixpoint iterations and widening operators for affine sets, along with some convergence results on fixpoint computations. They can be applied here component-wise to our abstract values.

Remark: computing the finite precision value only Simpler variations of this abstract domain can be used to compute the finite precision value only, basically using only one affine form. Each operation creates a new noise symbol that is used either to over-approximate a non-affine operation, either to add a rounding error, or both. The results for the finite precision value would be comparable to those we obtain through our more decomposed abstract domain, though some slight differences would be due to a different interpretation of tests and to the different decomposition (the results for the direct computation of the values would be slightly more accurate in general).

5 Implementation and results

These abstract domains are among those implemented in the Fluctuat analyzer, some of the case studies realized with the zonotopic abstract domain on real industrial programs, are reported in [17, 3, 5]. We discuss here the implementation in finite precision, and present benchmarks and some results on simple examples.

We now have at our disposal for the analysis, floating-point numbers with an arbitrary precision p, instead of real numbers (using the MPFR [6] library). For the non relational abstract domain, where operations on real numbers occur on interval bounds, the abstraction by operations with numbers in $\mathbb{F}_p$ is kept correct by using all such operations in interval arithmetic with outward rounding, as in (3) for the addition. The affine forms of the relational domain can also be computed using finite precision coefficients: the computation is made sound by over-approximating the rounding error committed on these coefficients and agglomerating the resulting error in new noise terms. Keeping track of these particular noise terms is interesting as they quantify the overestimation in the analysis specifically due to the use of finite precision for the analysis.

Finally, before eventually using a widening to ensure termination of the analysis, it proved interesting to use a convergence acceleration operator for the computation of fixpoint, obtained by the progressive reduction of the precision p of the floating-point used for the analysis (see [17], Section 2.4 for more details).

We now present some experiments made using Fluctuat, using both the non-relational and the relational abstract semantics; part of the programs (the academic ones) can be found at `http://www.lix.polytechnique.fr/~goubault/{NAME}[.c][.apron]`. Times are given on an Intel Core 2 Duo 2GHz, 4Gb of

RAM, running under MacOS Snow Leopard, and include all the mechanisms used in Fluctuat, including for instance alias analysis.

Let us first come back to the introductory example of Section 1: with the relational domain using 1000 subdivisions, in 78 seconds we get for the real value of t, $r^t \in [-2.10^{-6}, 2.10^{-6}]$ instead of $[-1.95, 1.94]$ without subdivision, and $[-8, 8]$ with the non-relational domain. We already saw that the rounding errors where negligible, we now also have that the two computations for y and z are functionally very close in real numbers.

Example 2. Take the following simple program that implements 100 iterations of a scheme proposed by Muller [23] and analyzed by Kahan, to demonstrate the effect of finite precision on computation.

```
1    x0 = 11/2.0;
2    x1 = 61/11.0;
3    for (i=1 ; i<=100 ; i++) {
4      x2 = 111 - (1130 - 3000/x0) / x1;
5      x0 = x1;  x1 = x2; }
```

Computed with exact numbers, this sequence should converge to 6. However, this fixed point is repulsive, while the fixed point 100 is attractive. This means that any perturbation from the exact sequence converging to this repulsive fixed point will eventually lead to the attractive one. Thus, when computed in finite precision, whatever the precision used, this sequence will eventually converge to 100. Fluctuat, with the interval abstract domain, and using floating-point numbers with 500 bits of precision, indeed finds in less than 0.15 second that $x2$ after the loop has a float value equal to 100, a real value equal to 5.999..., and a global error equal to $-94.000...$, due to lines 2 and 4, and to higher order errors. As a matter of fact, it is in particular because $x1$ is not represented exactly that the dynamical system converges towards the attractive fixpoint (in floating-point numbers). The fixed point for an arbitrary number of iterations will be $\top$ for the real value and error. Using APRON with Polka (polyhedra) and linearization, we find $x1$ equal to 5.999... in real-numbers and $x1$ equal to top (since the eighth iterate) in floating-point numbers, even though we unravelled all 100 iterations and infinite precision computation is made by the analyzer - this is due to the linearization scheme.

Example 3. The function below computes (Householder method) the inverse of the square root of the input I. The loop stops when the difference between two successive iterates is below a criterion that depends on a value `eps`.

```
  xn = 1.0/I; i = 0; residu = 2.0*eps;
  while (fabs(residu) > eps) {
    xnp1 = xn*(1.875+I*xn*xn*(-1.25+0.375*I*xn*xn));
    residu = 2.0*(xnp1-xn)/(xn+xnp1);
    xn = xnp1; i++; }
  O = 1.0/xnp1;  sbz = O- sqrt(I);
```

This algorithm is known to quickly converge when in real numbers, whatever the value of `eps` is. However, in finite precision, the stopping criterion must be defined with caution: for instance, executed in simple precision, with `eps`$=10^{-8}$, the program never terminates for some inputs, for instance $I = 16.000005$. Analyzed with Fluctuat for an input in $[16, 16.1]$, we obtain that the number of iterations i of the algorithm is potentially unbounded. We also obtain that $residu$ has a real value bounded in $[-3.66e^{-9}, 3.67e^{-9}]$ and a floating-point value is bounded in $[-2.79e^{-7}, 2.79e^{-7}]$ (50 subdivisions), so that the real value satisfies the stopping criterion but not the floating-point value, which is also signalled by an unstable test on the loop condition. Now, analyzing the same program but now for double precision variables, Fluctuat is able to prove in 3 seconds that the number of iterations is always 6 (both the real value and the floating-point value satisfy the stopping criterion). Also, bounding the variable `sbz`, Fluctuat is also able to bound the method error, so that we prove that the program indeed computes something that is close to the square root: the real and float value of `sbz` are found in $[-1.03e^{-8}, -1.03e^{-8}]$, with an error due to the use of finite precision in $[-1.05e^{-14}, 1.05e^{-14}]$. The non-relational analysis does not manage to bound the values of variables nor the number of iterations.

Example 4. Consider the following set of second-order linear recursive filters:

```
1   A1=[0.5,  0.8]; A2=[-1.5,-1]; A3=[0.8,1.3]; E0=[0,1.0];
2   B1=[1.39,1.41]; B2 =[-0.71,-0.69]; E=[0,1.0];
3   for (i=1;i<=N;i++) { E1=E0; E0=E; E=[0,1.0];
4     S1=S0; S0=S;
5     S=A1*E+E0*A2+E1*A3+S0*B1+S1*B2; }
```

We find in 90 seconds, with the relational abstraction (the non-relational abstraction does not converge): S in $[-15.08, 16.58]$ with error in $[-7.9.10^{-14}, 7.9.10^{-14}]$ (coming mostly from line 5). When unrolling the first 200 iterations, we find the better estimate (convergence is actually very fast): S in $[-5.63, 7.13]$, with error in $[-2.93.10^{-14}, 2.93.10^{-14}]$. Our abstraction allows for test case generation, both in values (see for instance [5]) and for errors: indeed, similarly as in [5], one can easily maximize or minimize the form $\hat{e}^x = e_0^x + \sum_i e_i^x \varepsilon_i^r + \sum_l e_l^x \varepsilon_l^e$ by choosing the right inputs, i.e. the values for ε_i^r (see Section 4.2). Here we find input values that allow for S to reach -0.92 and 5.36 (in real-numbers) and using the subdivision mechanism of Fluctuat [5], we find even better: [-4.81,6.33]. For errors, we find inputs reaching $-2.49.10^{-14}$ and $2.6.10^{-14}$ very close to the static analysis result.

Example 5. We finally consider a conjugate gradient algorithm applied to a class of matrices close to a Lagrangian in one dimension, discretized in a 4×4 matrix. Fluctuat shows that the error is coming from mostly `evalA`, the evaluation of the perturbed Lagrangian (error in $[-9.22.10^{-6}, 9.22.10^{-6}]$), the scalar product function (error in $[-6.61.10^{-6}, 6.61.10^{-6}]$), and the multiply add routine for matrices ($[-5.46.10^{-6}, 5.46.10^{-6}]$) for $xi[1]$. The influence of the perturbation of the Lagrangian matrix is negligible (of the order of $2.56.10^{-11}$), as well as the influence of the perturbation of the initial condition (of the order of $1.98.10^{-10}$):

this is a well-known phenomenon for well-behaved problems such as this one, called orthogonality defects, see [19].

In terms of performance (time and memory), the relational abstract domain is of the same order of complexity as the zonotopic domain for the analysis of real values (though of course a little more costly, as there will be the error noise symbols on top of the value noise symbols). For this abstract domain for the analysis of real values, benchmarks and comparisons with other classical abstract domains (APRON implementation) were presented in [8, 9].

Finally, here is a table of some experiments made using Fluctuat on real codes coming from industry, which involve some intricate numerical computations:

Program	#LOC	Analysis	Time(s)	Memory(Mo)
subset of navigation code	∼10000	fixpoint	33	20
physical process monitoring	∼800	fixpoint	188	94
subset of navigation code	∼1000	unfolding 20000 it	520	16

6 Conclusion and future work

We presented in this paper non-relational and relational abstractions of finite-precision computations allowing for determining precisely not only the bounds of variables in real-number and finite precision (in particular, floating-point number) semantics, but also the discrepancy between these two semantics, and their causes. The amount of information delivered is more important than in a classical floating-point analysis, however we can obtain better results, in a very economical manner, both in time and memory. This extra information also allows us to generate interesting outcomes such as worst-case inputs.

As a last word, let us notice that the rounding error of (normalized) floating-point computations is classically bounded in relative error. In a relational analysis, we would thus like to express the new rounding error of a real number defined by an affine form over noise symbols, as a function of these noise symbols. However, this is no longer expressible purely as an affine form, hence we simply use a noise symbol to abstract the whole range of error. A future direction could be to enhance the abstract domain with the relative error, when particularizing our technique to floating-point arithmetic.

References

1. S. Boldo and J.-C. Filliâtre. Formal Verification of Floating-Point Programs. In *18th IEEE International Symposium on Computer Arithmetic*, June 2007.
2. J. L. D. Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics. *SEBGRAPI'93*, 1993.
3. E. Conquet, P. Cousot, R. Cousot, E. Goubault, K. Ghorbal, D. Lesens, S. Putot, and M. Turin. Space software validation using abstract interpretation. In *Proceedings of DASIA*, 2009.
4. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

5. D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In *14th FMICS*, volume 5825 of *LNCS*, 2009.
6. P. Zimmerman et al. The MPFR library. available at http://www.mpfr.org/.
7. IEEE 754 Standard for Binary Floating-Point Arithmetic. 2008 revision, http://ieeexplore.ieee.org/servlet/opac?punumber=4610933.
8. K. Ghorbal, E. Goubault, and S. Putot. The zonotope abstract domain taylor1+. In *CAV*, volume 5643 of *LNCS*, 2009.
9. K. Ghorbal, E. Goubault, and S. Putot. A logical product approach to zonotope intersection. In *22nd CAV*, volume 6174 of *LNCS*, 2010.
10. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.
11. E. Goubault. Static analyses of the precision of floating-point operations. In *8th Static Analysis International Symposium SAS'01*, volume 2126 of *Lecture Notes in Computer Science*, pages 234–259, 2001.
12. E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: A simple abstract interpreter. In *11th European Symposium on Programming ESOP'02*, volume 2305 of *Lecture Notes in Computer Science*, pages 209–212, 2002.
13. E. Goubault and S. Putot. Weakly relational domains for the analysis of floating-point computations. Presented at NSAD, 2005.
14. E. Goubault and S. Putot. Static analysis of numerical algorithms. In *SAS'06*, volume 4134 of *LNCS*, 2006.
15. E. Goubault and S. Putot. Perturbed affine arithmetic for invariant computation in numerical program analysis. *CoRR*, abs/0807.2961, 2008.
16. E. Goubault and S. Putot. A zonotopic framework for functional abstractions. *CoRR*, abs/0910.1763, available at http://arxiv.org/abs/0910.1763, 2009.
17. E. Goubault, S. Putot, P. Baufreton, and J. Gassino. Static analysis of the accuracy in control systems: Principles and experiments. In *12th FMICS*, volume 4916 of *LNCS*, 2007.
18. M. Martel. Propagation of roundoff errors in finite precision computations: A semantics approach. In *11th ESOP*, volume 2305 of *LNCS*, 2002.
19. G. Meurant. *The Lanczos and Conjugate Gradient Algorithm; from theory to finite precision computation*. SIAM, 2006.
20. A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP*, volume 2986 of *LNCS*, 2004.
21. A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *VMCAI'06*, pages 348–363, 2006.
22. D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):1–41, 2008.
23. J.-M. Muller. *Arithmétique des Ordinateurs*. Masson, 1989.
24. J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.
25. APRON Project. Numerical abstract domain library, 2007. `http://apron.cri.ensmp.fr`.
26. S. Putot, E. Goubault, and M. Martel. Static analysis-based validation of floating-point computations. In *Numerical Software with Result Verification*, volume 2991 of *LNCS*, 2003.
27. R. Skeel. Roundoff error and the patriot missile, 1992. SIAM News.
28. P. H. Sterbenz. Floating point computation. Prentice Hall, 1974.