# Static Analysis of the Accuracy in Control Systems: Principles and Experiments

Eric Goubault[1], Sylvie Putot[1], Philippe Baufreton[2], and Jean Gassino[3]

[1] CEA, LIST, Boîte courrier 65, GIF-SUR-YVETTE CEDEX, F-91191 France
{eric.goubault,sylvie.putot}@cea.fr
[2] Hispano-Suiza
philippe.baufreton@hispano-suiza-sa.com
[3] IRSN
jean.gassino@irsn.fr

**Abstract.** Finite precision computations can severely affect the accuracy of computed solutions. We present a complete survey of a static analysis based on abstract interpretation, and a prototype implementing this analysis for C code, for studying the propagation of rounding errors occurring at every intermediary step in floating-point computations. In the first part of this paper, we briefly present the domains and techniques used in the implemented analyzer, called FLUCTUAT. We describe in the second part, the experiments made on real industrial codes, at Institut de Radioprotection et de Sûreté Nucléaire and at Hispano-Suiza, respectively coming from the nuclear industry and from aeronautics industry. This paper aims at filling in the gaps between some theoretical aspects of the static analysis of floating-point computations that have been described in [13,14,21], and the necessary choices of algorithms and implementation, in accordance with practical motivations drawn from real industrial cases.

**Keywords:** Static analysis, floating-point computations, control systems.

## 1 Introduction

The use of floating-point arithmetic as a computable approximation of real arithmetic may introduce rounding errors at each arithmetic operation in a computation. Even though each of these errors is very small, their propagation in further computations, for example in loops, can produce a dramatic imprecision on a critical output. We propose an analysis that computes for each variable an over-approximation of the difference between a same computation in real and in floating-point arithmetic. Moreover, we decompose this difference over the operations that introduced errors, thus pointing out the operations responsible for the main losses.

*Principal contributions.* This paper essentially surveys the different abstract interpretation domains and techniques used in the static analyzer FLUCTUAT. The emphasis is not put on the theoretical details of the main domain used for

S. Leue and P. Merino (Eds.): FMICS 2007, LNCS 4916, pp. 3–20, 2008.

abstracting floating-point values, which can be found in [14]. But it describes the main techniques and answers to the difficulties we had to address in the practical use of these domains in an analyzer. Also, a large part is dedicated to some examples of the analysis which was led on industrial examples.

*Related work.* Few complete static analyzers of C programs are fully described in the literature. This paper tries to match some of the available descriptions of the ASTREE analyzer [1,2] which is probably the most complete analyzer today, with respect to the number of techniques and domains implemented, in particular. Also, some commercial abstract interpreters like CodeSonar [24] and PolySpace [25] are now available. All these tools analyze run-time errors mostly, whereas we analyze a subtle numerical property, namely the discrepancy introduced by the use of floating-point numbers instead of real numbers, in C programs. This requires very fine and specific abstract domains, and the difficulty lies mostly in the numerical subtleties of small parts of a code, and not in the size of the program.

*Overview.* This paper is divided in two parts. In Section 2, we present the abstract domains and main techniques used in the FLUCTUAT analyzer. In subsection 2.1, we briefly introduce the abstract domain for representing floating point variables, which is described in more details in [14]. Then in subsection 2.2, we detail how this domain is extended to integer variables, and the specificities and difficulties of handling integers. We then describe in subsection 2.3, the simple aliasing model we are using, when it comes to abstract pointers, structures and arrays. The iteration strategy which is used to solve the semantic equations (and in particular specific widening operators), is described in subsection 2.4. Finally, some assertions in a language specific to the analyzer, allow us to specify properties of variables, such as set of possible input values, but also more subtle properties such as bounds on the gradient of an input over iterations in a loop. These are presented in subsection 2.5.

We then discuss in Section 3 some experiments conducted with FLUCTUAT on industrial codes. We first describe the analysis of some programs developed at Hispano-Suiza in the aeronautics industry. We first concentrate on some interesting specific sub-functions, and then come to a full control application. In a second part, we describe the analysis of a code from the nuclear industry that IRSN has to expertise.

## 2 Abstract Domains and Techniques Used in FLUCTUAT

### 2.1 Floating-Point Variables

**General principles.** The analysis bounds at each operation the error committed between the floating-point and the real result. It relies for this on a model of the difference between the result $x$ of a computation in real numbers, and the result $f^x$ of the same computation using floating-point numbers, expressed as a sum of error terms

$$x = f^x + \bigoplus_{\ell \in L \cup \{hi\}} \omega^x_\ell \, . \qquad (1)$$

We assume that a control point of a program is annotated by a unique label $\ell \in L$. In this relation, a term $\omega_\ell^x$, $\ell \in L$, denotes the contribution to the global error of the first-order error introduced by the operation labeled $\ell$. It expresses both the rounding error committed at label $\ell$, and its propagation due to further computations on variable $x$. Errors of order higher than one, coming from non-affine operations, are grouped in one term associated to special label $hi$. We let $\mathcal{L}$ be the union of $L$ and $hi$.

Let $\mathbb{F}$ be either the set of single or double precision floating-point numbers. Let $\uparrow_\circ : \mathbb{R} \to \mathbb{F}$ be the function that returns the rounded value of a real number $r$, with respect to the rounding mode $\circ$. The function $\downarrow_\circ : \mathbb{R} \to \mathbb{F}$ that returns the roundoff error is then defined for all $f \in \mathbb{R}$, by $\downarrow_\circ (f) = f - \uparrow_\circ (f)$. The result of an arithmetic operation $\Diamond^{\ell_i}$ contains the combination of existing errors on the operands, plus a new roundoff error term $\downarrow_\circ (f^x \Diamond f^y)\varepsilon_{\ell_i}$. For addition and subtraction, the existing errors are added or subtracted componentwise, and a new error $\downarrow_\circ (f^x + f^y)$ is associated to control point $\ell_i$:

$$x +^{\ell_i} y = \uparrow_\circ (f^x + f^y) + \bigoplus_{\ell \in \mathcal{L}} (\omega_\ell^x + \omega_\ell^y) + \downarrow_\circ (f^x + f^y) \ .$$

The multiplication introduces higher order errors, we write:

$$x \times^{\ell_i} y = \uparrow_\circ (f^x f^y) + \bigoplus_{\ell \in \mathcal{L}} (f^x \omega_\ell^y + f^y \omega_\ell^x) + \sum_{(\ell_1, \ell_2) \in \mathcal{L}^2} \omega_{\ell_1}^x \omega_{\ell_2}^y + \downarrow_\circ (f^x f^y) \ .$$

We refer the reader to [10,19] for more details on this domain.

**Unstable tests.** Our approach is that of abstract interpretation [7], and all control flows due to sets of possible inputs are considered. But there is one specific difficulty due to floating-point computations. Indeed, in tests, the branch followed by the floating-point and the corresponding real value of a variable can be different, we then call them unstable tests (as in [26]). Consider for example the following portion of code, supposing input $x$ is in interval [1,3] with an error equal to 1.0e-5:

```
if (x <= 2) x = x+2;
```

Then, for $x$ equal to 2 for example, the floating-point result after this test is 4, whereas the result of this program if it were executed on the real semantics would be 2.00001. But handling this divergence in control flow in the general case would be complicated and costly, and quickly very imprecise. For example here, if we consider the different control flows, we find the floating-point value of $x$ in $[2, 4]$, with an error in interval $[1.0e^{-5}, 2]$. Without any additional relation between values and errors, this result is highly imprecise.

We thus made the choice in the Fluctuat analyzer to make the assumption that the real and floating-point flows take the same branches. The result given here at the end of the program would thus be $x = [2, 4]$ with an error equal to $1.0e^{-5}$ (if we neglect the additional rounding error due to the addition).

However, when the analyzer detects, as is the case here, that the control flows may be different, it issues a warning.

**Relational domain.** A natural abstraction of the coefficients in expression (1), is obtained using intervals. The machine number $f^x$ is abstracted by an interval of floating-point numbers, each bound rounded to the nearest value in the type of variable $x$. The error terms $\omega_i^x \in \mathbb{R}$ are abstracted by intervals of higher-precision numbers, with outward rounding. However, results with this abstraction suffer from the over-estimation problem of interval methods. If the arguments of an operation are correlated, the interval computed with interval arithmetic may be significantly wider than the actual range of the result.

We thus proposed and implemented a relational domain, relying on affine arithmetic [5,22] for the computation of the floating-point value $f^x$. Affine arithmetic uses affine correlation between real variables, and allows us to get much tighter results than classical interval arithmetic (the concretisation forms zonotopes: center-symmetric bounded convex polytopes). It relies on a representation of a quantity $x$ by an affine form, which is a polynomial of degree one in a set of noise terms $\varepsilon_i$:

$$\hat{x} = \alpha_0^x + \alpha_1^x \varepsilon_1 + \ldots + \alpha_n^x \varepsilon_n, \text{ with } \varepsilon_i \in [-1, 1] \text{ and } \alpha_i^x \in \mathbb{R}. \tag{2}$$

Each noise symbol $\varepsilon_i$ stands for an independent component of the total uncertainty on the quantity $x$, its value is unknown but bounded in [-1,1]; the corresponding coefficient $\alpha_i^x$ is a known real value, which gives the magnitude of that component. The sharing of noise symbols between variables expresses implicit dependencies. The full semantics is described in [14], and linearizes floating-point expressions dynamically (and not statically as in [20]). The semantics is memory-efficient: it needs only a small factor of the size that an (economic) interval analysis would take. No a priori decided packing of variables [2] is needed since the representation of relations is implicit [14]. Nevertheless, we use a sparse representation of the global environment, akin to the one described in Section 5 of [1].

The coefficients $\alpha_i^x$ have no meaning relevant to our analysis, the decomposition is a mean for a more accurate computation. This is different from expression (1), where coefficient $\omega_\ell^x$ represent the contribution of control point $\ell$ to the total rounding error. However, they can be used for an analysis of the sensibility of a program to an input: when an input is taken in a small interval, a new noise symbol $\varepsilon_i$ is created. The evolution of the corresponding $\alpha_i^x$ in further computations indicates how this initial uncertainty is amplified or reduced [23].

These affine forms allow us to represent results of real arithmetic. The analysis must be adapted to the case of floating-point arithmetic, where symbolic relations true in real arithmetic do no longer hold exactly. We thus decompose the floating-point value $f^x$ of a variable $x$ resulting from a trace of operations, in the real value of this trace of operations $r^x$, plus the sum of errors $\delta^x$ accumulated along the computation, $f^x = r^x + \delta^x$. The real part is computed using affine arithmetic, and the error is computed using three intervals that respectively bound the error on the lower and upper bounds of the set of real values $r^x$, and the maximum error on all this set. Without going into too much detail, we can say that these errors on bounds allow us to improve the estimates for the floating-point bounds, compared to using the maximum error. But the maximum error is still needed at each step to estimate the results of further computations.

This domain for the values of variables, is of course more expensive than interval arithmetic, but comparable to the domain used for the errors. And it allows us to accurately analyze non trivial numerical computations, as we will show in Section 3. We also plan to introduce a relational computation for errors, along the lines of [16]. First ideas on these relational semantics were proposed in [13], [21]. The relational semantics for the value $f^x$ is described in detail in [14], with in particular the lattice operations such as join and meet.

### 2.2 Integer Variables

We implemented modular integer arithmetic semantics, and a domain consisting of value (coded by affine forms) plus sum of errors is used as for floating-point variables. For example, when adding one to the greatest integer that can be represented in the `int` type, say `INT_MAX`, the value of the result is the smallest integer represented by an `int`, say `INT_MIN`, and an error of `INT_MAX-INT_MIN+1` is associated to this variable. Conversions between integers and floating-points are supported, and the errors are propagated.

**Bitwise operations.** Some attention must be paid to the propagation of errors on operands in order to avoid losing too much precision. Indeed, the behavior of the `and`, `or` and `xor` operators is non affine with respect to the operands. In the general case, the errors on the operands $x$ and $y$ are propagated as follows:

- we compute the result of $\diamond$ on the sets of floating-point values, $f^z = f^x \diamond f^y$,
- we compute the result of the same operation on the interval bounds for the real values, $r^z = r^x \diamond r^y$, with $r^x = f^x + \sum_l \omega_l^x$ and $r^y = f^y + \sum_l \omega_l^y$,
- then the propagated error on $z$ is $r^z - f^z$, and it is associated to the label of the current operation.

There are two consequences. First, we lose the decomposition of errors on operations executed before bitwise operations. Second, the larger the intervals $f^x$ and $f^y$, the more over-approximated the propagated errors are. We thus propose an option of the analyzer to locally subdivide one of these intervals in the propagation of errors: the cost of a bitwise operation is approximately multiplied by the number of subdivisions, but this cost is in general negligible compared to the full analysis, and the results can be greatly improved.

Error terms are agglomerated for the same reason for the division and modulo operators on integers. The error is also computed as the difference between the floating-point interval result and the real interval result: local subdivisions can again greatly improve the estimation of errors.

**Conversions.** The semantics for the conversion between floating-point numbers, and with integers differ by the meaning we give to each:

- in the conversion from double precision to single precision floating-point numbers, we consider the difference between the initial double precision value and the result of the conversion, as an error on the result.
- in the conversion from a floating-point number to an integer, we consider that the truncation is wanted by the user, and is thus not an error. A new error can be introduced by such a conversion only when the floating-point number exceeds the capacity of the integer type. However, all errors are grouped in one integer term corresponding to the label of the conversion.
- in the conversion from an integer to a floating-point number, most of the time no precision is lost, and the sum value plus errors is transmitted as is. However, this is not always the case, and an error still has to be added in some cases: for example a 32 bits integer with all bits equal to 1 cannot exactly represented by a single precision floating-point number, which mantissa is represented on 23 bits.

We encountered some other cast operations that we included in the set of instructions understood by the analyzer, such as the ones used to decode and encode IEEE 754 format, directly by bitwise operations. Take for instance the following piece of code (assuming 64 bits little endian encoding for `double`):

```
double Var = ...; signed int *PtrVar = (signed int *) (&Var);
int Exp = (signed int) ((PtrVar[0] & 0x7FF00000) >> 20) - 1023;
```

We cast variable `Var` into an array of 32 bits types. Then we extract the first 32 bits of the 64 bits word. The rest of the manipulation of the program above, masks the bits of the mantissa, and shifts the value, to get in `Exp` the binary exponent in IEEE754 format of the value stored in `Var`.

In the interpretation of this case by FLUCTUAT, all error terms are agglomerated in one corresponding to the label of the cast, and local subdivisions of the values can be applied to improve the bounds for the errors, as for bitwise operations.

### 2.3 Aliases and Arrays

Our alias and array analysis is based on a simple points-to analysis, like the ones of [18], or location-based alias analyses. An abstract element is a graph of locations, where arcs represent the points-to relations, with a label (which we call a selector) indicating which dereferencing operation can be used. Arrays are interpreted in two different ways, as already suggested by some of the authors in [9]: all entries are considered to be separate variables (called "expanded" in [1]) or the same one (for which the union of all possible values is taken - called "smash" in [1]). These abstractions have proven sufficient typically for SCADE generated C programs.

### 2.4 Iteration Strategy

**Loops.** The difficulty in loops is to get a good over-approximation of the least fixpoint without too many iterations of the analyzer. For that, we had to design adapted iteration strategies:

- in the case of nested loops, a depth first strategy was chosen: at each iteration of the outer loop, the fixpoint of the inner loop is computed,
- a loop is unfolded a number of times (similar to the "semantic loop unrolling" of [1]), before starting Kleene iterations (unions over iterations),
- some particularities of our domain require special care in loops: for example, noise symbols are potentially introduced at most operations, and there are new noise symbols for each iteration of a loop. But we can choose to reduce the level of correlation we want to keep, and for example keep correlations only between the last $n$ iterations of a loop, where $n$ is a parameter of the analyzer. Also, we can choose to agglomerate or not some noise symbols introduced in a loop when getting out of it. This allows us to reduce the cost of the analysis while keeping accurate results.
- acceleration techniques (widenings) adapted to our domain had to be designed. In particular, widenings are not always performed at the same time on integer or floating-point variables, and on values or error terms. Also, we have designed a widening specially adapted to floating-point numbers, by gradually reducing the precision of the numbers used to bound the terms: this accelerates the convergence of Kleene iteration compared to iteration with fixed precision, and allows us to get very accurate results. This should be thought of as an improved method than the "staged widening with thresholds" of [1], in the sense that thresholds are dynamically chosen along the iterations, depending on the current values of the iterates. After a number of these iterations, a standard widening is used.

To illustrate this last point (progressive widening by reduction of the precision), let us consider the fixpoint computation of

```
while () x = 0.1*x;
```

with no unrolling of the loop, starting from $x_0 \in [0,1]$. With our simple (non relational) semantics, we have, with $ulp(1)$ denoting the machine rounding error around 1,

$$\begin{aligned} x_1 &= [0,1] + \delta\varepsilon_2,\;\; \delta = 0.1[-ulp(1), ulp(1)] \\ x_2 &= [0,1] + (0.1\delta + \delta)\varepsilon_2 \\ x_n &= [0,1] + (\sum_{k=0}^{n} 0.1^k)\delta\varepsilon_2. \end{aligned}$$

If real numbers are used to compute the error term, without any widening the computation does not terminate even though the error term remains finite. Now if floating point numbers are used to bound the error term, the convergence depends on the number of bits used to represent the mantissa. For simplicity's sake, let us consider $\delta = [-1,1]$, and radix 10 numbers. With 3 significant digits, a fixpoint is got in 4 iterations:

$$\begin{aligned} &\omega_1 = \delta = [-1,1] \\ &\omega_2 = \uparrow_\infty ([-0.1, 0.1] + [-1,1]) = [-1.1, 1.1] \end{aligned}$$

$$\omega_3 = \uparrow_\infty ([-0.11, 0.11] + [-1, 1]) = [-1.11, 1.11]$$
$$\omega_4 = \uparrow_\infty ([-0.111, 0.111] + [-1, 1]) = \uparrow_\infty [-1.111, 1.111] = [-1.12, 1.12]$$
$$\omega_5 = \uparrow_\infty ([-0.112, 0.112] + [-1, 1]) = \uparrow_\infty [-1.112, 1.112] = [-1.12, 1.12]$$

More generally, we can show that with N significant digits, a fixpoint is got in N+1 iterates. Thus reducing the precision of numbers accelerates the convergence towards a (larger) fixpoint.

Of course, this is a toy example, in practice the fixpoint is computed by unrolling the loop a certain number of times before beginning the unions, which here solves the problem. But we are confronted with this kind of computations in the general case. And in more complicated examples, when the optimal unrolling was not chosen, this widening allows us to still compute an interesting fixpoint.

**Interprocedural analysis.** In critical embedded systems, recursion is in general prohibited. Hence we chose to use a very simple interprocedural domain, with static partitioning, based on [17].

### 2.5 Assertions

A number of assertions can be added to the analyzed C source code, to specify the behavior of some variables of the program. For a single precision floating-point variable x, the assertion:

```
x = __BUILTIN_DAED_FBETWEEN(a,b);
```

indicates that x can take any floating-point value in the interval $[a, b]$. The same assertions exist to define the range of double precision or integer variables.

One can also specify an initial error together with the range of values for a variable. For example,

```
x = __BUILTIN_DAED_FLOAT_WITH_ERROR(a,b,c,d);
```

specifies that variable x of type float takes its value in the interval $[a, b]$, and that is has an initial error in the interval $[c, d]$.

In some cases, bounds on the values are not sufficient to describe accurately the behavior of a system: we thus added an assertion that allows us to bound, in a loop indexed by an integer variable i, the variation between two successive values of an input variable x:

```
for (i=i0 ; i<N ; i++)
x = __BUILTIN_DAED_FGRADIENT(x0min,x0max,gmin(i),gmax(i),xmin,xmax,i,i0);
```

In this assertion, $i_0$ is the value of variable $i$ at first iteration. The value of x at first iteration is in the interval [x0min,x0max], the difference between two successive iterates is in the interval [gmin(i),gmax(i)], which bounds may depend on the iterate, and the value of x is always bounded by [xmin,xmax]. Thus $x(i_0)$ =[x0min,x0max], and for all $i \in \{i_0, \dots, N\}$, we have $x(i) = (x(i-1)$+[gmin(i),gmax(i)]) $\bigcap$[xmin,xmax]. Our relational domain (subsection 2.1) is specially well adapted to dealing with these relations between iterates in a loop. An example of the use of this assertion is given in the worst-case scenario part of example presented in subsection 3.2.

**Subdivisions of inputs.** In the example `SqrtR` of subsection 3.1, even with the relational domain, non-linearities of the studied iterative scheme produce too much imprecision, and the solver of the abstract equations does not prove the termination of the analyzed algorithm. A solution to this is to restrict the range of values of the inputs, for which we want to analyze the program, so that we are close enough to linear behaviors. This is done in FLUCTUAT by subdividing the domain of some inputs whose ranges are already bounded by assertions of the type `__BUILTIN_DAED_FBETWEEN`. The user can select one or two such variables to be subdivided by pointing in the program the corresponding assertions.

FLUCTUAT analyzes independently the program as many times as we subdivide some of the inputs. Suppose we subdivide $n$ times an input variable $x$ which has range, defined by an assertion, in $[a, b]$: the analyzer will analyze the program with $x$ in $[a, a+\frac{b-a}{n}]$, then in $[a+\frac{b-a}{n}, a+2\frac{b-a}{n}]$, ..., $[a+(n-1)\frac{b-a}{n}, b]$. Hence it does not need more memory than needed for one analysis, but takes about $n$ times the duration of one analysis, where $n$ is the number of subdivisions. In the case when we subdivide two such assertions, the subdivisions are completely independent, hence leading to a quadratic factor time increase of the analysis. We chose not to offer the user the possibility to subdivide the values of more than two input variables because it would lead to too slow analyses. This would become reasonable only for parallel versions of FLUCTUAT.

This kind of subdivision cannot be used for an assertion in a loop, because it would be equivalent to choosing at all iterates of the loop the values of $x$ to be in the same sub-interval. Indeed, subdividing independently all iterates would be far too costly, and maybe not either what is really intended by the user. We thus proposed the special assertion `__BUILTIN_DAED_FGRADIENT` for these cases of reactive programs, where inputs are acquired cyclically over time.

# 3 Experiments on Control Systems

## 3.1 Hispano-Suiza

The Full Authority Digital Engine Control, better known as a FADEC, is one of the largest electronic control units on an aircraft. It continuously processes and analyzes key engine parameters, to make sure the engine operates at maximum potential. The following test cases for FLUCTUAT are extracted from pieces of code which have been written during the development of reusable libraries, designed for the FADEC. They are representative of the code of the FADEC, and some of them present some hard numerical difficulties for static analyzers.

**Experiments on elementary symbols.** We examined several elementary symbols used in applications at Hispano-Suiza. Elementary symbols are manually developed and coded independently from SCADE which is used as a design tool. Among these symbols was the following code (slightly changed for convenience), which is intended to return `Output` equal to the square root of `Input` by a Householder method.

```
void SqrtR (double Input)
{   double xn, xnp1, residu, _EPS, Output;
    int i, cond;
    _EPS = 0.000001;
    Input = __BUILTIN_DAED_DOUBLE_WITH_ERROR(0.1,20.0,0,0);
    if (Input <= 1.0)
      xn = 1.0;
    else
      xn = 1.0/Input;
    xnp1 = xn;    residu = 2.0*_EPS*(xn+xnp1);
    i = 0;
    while (fabs(residu) > _EPS * (xn+xnp1))
    {
      xnp1 = xn * (1.875 + Input*xn*xn*(-1.25+0.375*Input*xn*xn));
      residu = 2.0*(xnp1-xn);
      xn = xnp1;
      i++;
    }
    Output = 1.0 / xnp1;
    should_be_zero = Output*Output - Input;
}
```

This involves the iteration, until some residue is less than a small value _EPS=$e^{-6}$, of a fifth-order polynomial. *The number of iterates for the algorithm to converge is thus not given by the syntax of the program and must be the result of an accurate analysis.* Also, to this short program was added a last computation `should_be_zero = Output*Output - Input;` that enables a functional proof of the algorithm. Indeed, if variable `should_be_zero` is proved (both as a real and as a floating-point number) to be close to zero, this proves that the algorithm really computes something that is close to the square root of `Input`, for all values of `Input` in the given range [0.1,20].

When `Input` is in $[0.1, 20]$ as above, FLUCTUAT with 100000 subdivisions converges to a finite and precise estimate of the floating-point value of `Output` and of the number of iterates of the studied algorithm. Indeed, it finds `Output` to be in $[3.16e^{-1}, 4.48]$ with global error in $[-2.56e^{-13}, 2.56e^{-13}]$. And the number of iterates in the main loop `i` is found to be within 1 and 6 for the floating-point version of the Householder algorithm. This number of iterates is an exact result, as can be confirmed by using FLUCTUAT in the symbolic execution mode for `Input` equal to 1 and to 20 respectively (or alternatively, by checking the execution of the real binary file). Also, the analyzer reports an unstable test for the stopping condition of the loop. Indeed, the number of iterations of the Householder algorithm before convergence may be different with the floating-point semantics and with the real semantics! And if we took constant `_EPS` too small, the algorithm in real numbers would still terminate, while the analyzer would prove that the floating-point algorithm does not always terminate (see

[12] for more details). Note that no other static analyzer we know of is able to find even a good approximation of the floating-point enclosure of `Output`.

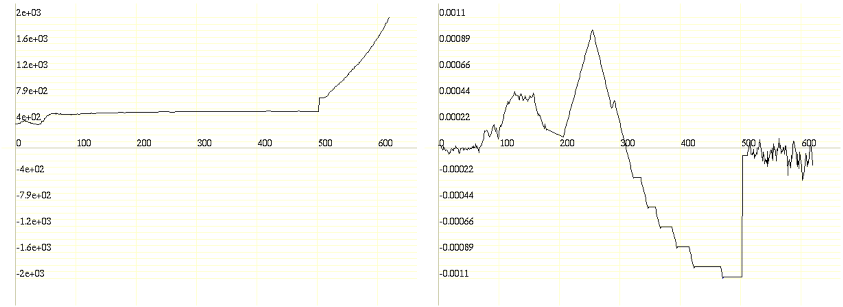
When perturbing the input by an error, for example in the range $[-e^{-8}, e^{-8}]$ and still subdividing 100000 times, we find in 603 seconds and 4Mb of memory on a 1 Ghz laptop PC, the same floating-point enclosure, and a global error in $[-3.06e^{-6}, 3.06e^{-6}]$ which is mainly due to the initial error of the order of $e^{-8}$. This shows the good behavior of the algorithm. Even though the results on the global error seem satisfying, we hope to be able to improve them a lot with a new relational domain on the error terms, as sketched in [13].

In fact, FLUCTUAT does not need to subdivide equally for all ranges of the input. For instance, with `Input` restricted to [16,20], it needs only 133 subdivisions to converge. Whereas, with `Input` restricted to [0.1,1], it needs about 4500 subdivisions. Hence a dynamic subdivision mode is planned for a future version.

Finally, let us consider the fonctional proof of the algorithm: that is, we no longer consider only the relevance of the floating-point implementation compared to the scheme in real number, but also measure the quality of the approximation of the square root by this algorithm. In order to get good results here, we need the fully relational version of Fluctuat which is only partially implemented. We thus considered only a reduced range $[16, 20]$ for Input, so that a fixed number of 6 iterations for the Householder algorithm corresponds to the number of iterations needed to reach the stopping criterion of the loop for this range of Input. We then prove (with subdivisions of Input), that the real value of `should_be_zero` is in the range $[-4.e^{-15}, 4.e^{-15}]$. This allows us to conclude that the algorithm, for this range of Input, computes in real number a result which is not further from the actual square root than $6.5e^{-8}$. The analyzer also bounds the error between the real and the floating-point value of `should_be_zero` in $[-5.e^{-14}, 5.e^{-14}]$. Then the approximation of the square root actually computed is not further from the result of the algorithm in real number than $2.5e^{-07}$. In fact, with more subdivisions, we could prove that the real value is even smaller, but the computation time becomes unreasonable for such a small program. However, the computation of the error remains stable when we subdivide the input, which leads us to think that it may be a tight over-approximation of the error.

These results all indicate a good behavior of the algorithm for `Input` in the range $[0.1, 20]$. However, this function was designed to be used for `Input` in $[1e^{-50}, 1e^{50}]$. Symbolic execution shows that the algorithm is much less satisfying for this extended range, and may need up to 95 iterations, which is too large for practical use, because of timing constraints. Since then, the algorithm for the square root has been changed.

**Representative code.** The following test case for FLUCTUAT is extracted from pieces of SCADE code which have been generated during the development of a military engine controller. The control law named `asservxn2` is aimed to control the speed of the Low-Pressure Compressor. Therefore, the control loop should be stable inside the whole flight domain, including the fuel flow `wf32cb` and the motor regime `xn2`. The code is 2358 lines long in C (44 functions, among which filters, interpolators, integrators), uses complex nested compound



Value (float) Global error

**Fig. 1.** Evolution of the fuel flow `wf32cb` for a given target motor regime

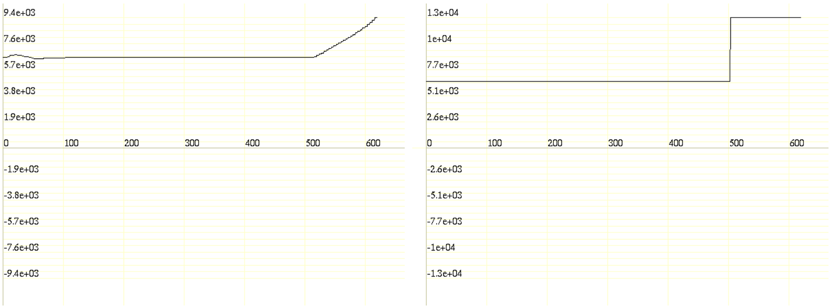
structures containing arrays and pointers, and has 167 integer variables and 269 floating-point variables known at the end of the main function (and many more local variables).

We have first used the static analyzer as an abstract interpreter (hence delivering information about precision loss) on the test scenarios that have been used to certify the program. As an example, the first test scenario consists in showing that the rotation regime `xn2` is well controlled by the command on the flow `wf32cb`. FLUCTUAT has been run in symbolic execution mode (i.e. with the semantics described in Section 2.1, but on one control flow only) on the sequence of 2500 consecutive inputs, on a 50 seconds duration. The scenario corresponds to a target low pressure regime shown in Figure 2 for the first 650 inputs. The control program computes the fuel flow necessary to reach this regime, see Figure 1. As shown in the excerpt of the test scenario, the motor regime is well controlled by `wf32cb`: when `wf32cb` increases, `xn2` increases as well until it reaches its target value `xn2cs`, in which case `wf32cb` stabilizes. At iteration 500, the target regime is increased and the control begins. The error in the command `wf32cb` is shown to be always bounded by $10^{-3}$ in absolute value, which indicates a good (relative) precision of the control algorithm.

Other similar tests have been carried out. We are in the process of studying the code for more general inputs (i.e. for ranges of target motor regimes), in a similar manner as done in next section (using gradient constraints on the inputs). It is to be noted though that the control mechanism uses an integrator, which is known to be hard to analyze, see for instance [12].

### 3.2 Institut de Radioprotection et Sûreté Nucléaire (IRSN)

Computer systems are increasingly used for safety purposes in nuclear reactors. For example, on the latest French power reactor series, software is used to perform safety functions including critical ones like protection. The Protection System does not control the process but monitors it, by continuously acquiring parameters like water pressure, temperatures in different pipes, neutron flux, and so on. From these inputs, the system computes tens of values using classical



`xn2` `xn2cs`

**Fig. 2.** Evolution of the motor regime `xn2`, and of its target `xn2cs`

data processing techniques: filters, arithmetic and logic operations, non-linear functions, thresholds. The system then checks that these computed values remain within the authorized domain. If not, it has to automatically shutdown the reactor within half a second, and to trigger safety systems like water injection or spraying, depending on the nature of the incident.

The following test case (mean-square filter), was taken from a representative piece of code that IRSN has to give expertise on, for the French nuclear certification body. Several key process parameters are sampled every 50 milliseconds by the protection system, which stops the reactor if a given threshold is exceeded. Unfortunately, the readings are affected by noise, which could delay a necessary stop or, on the contrary, cause a spurious one. Simply reducing the actual power of the plant below the threshold to provide a noise margin is not adequate for obvious economic reasons. On the other hand a spurious stop is also undesirable because it induces strong promptings to the mechanical structures and prevents the plant to produce electricity for several hours. Thus, a least-square linear regression filter is applied to improve the estimate of the most sensitive parameter. It is important however to make sure that this filtering step does not add too much numerical error due to rounding, we thus used FLUCTUAT to bound the error committed in it, and study the propagation of existing errors.

The filter is adaptative, that means its depth $D_k$ can vary at each cycle $k$, according to a formula depending on the signal value. The input sample $(i_k)_k$ is in an interval $[1e^2, 1.5e^8]$, and is transformed to give the input of the filter, by $Y_k = \log(ai_k + b)$. In this transformation, only ranges are known for parameters $a$ and $b$, with nominal values.

**Worst-case scenario.** We first consider a reduced version of the filter, using the fact that the filter can be written in such a way that outputs are independent, except that the depth of the filter depends on the previous values. It can be shown that, with the parameters used, the depth of the filter is always bounded and the inputs are in the maximum possible range $Y_k \in [8.42, 22.4]$. We thus study a worst-case scenario, that allows us to get bounds for the values and errors on the output that hold true for any step of the filter.

- We first suppose all inputs are independent and can take any value in this range at any step, using assertion `Yk = __BUILTIN_DAED_FBETWEEN(8.42,22.4);` Then we get with the relational domain, the following enclosures of the filtered value O, and of a value S related to the variation speed:

$$
\begin{aligned}
O &= [3.912, 26.907] + [-1.91e^{-4}, 1.91e^{-4}]\varepsilon \\
S &= [-0.350, 0.350] + [-5.37e^{-6}, 5.37e^{-6}]\varepsilon
\end{aligned}
$$

In these two expressions, the first interval bounds the floating-point value, the second one bounds the rounding error, the filter being implemented using single precision floating-point numbers.

- In order to have a more representative model of the inputs, we then used the assertion on the gradient to limit the variation between two successive inputs: we still take the range of inputs equal to [8.42, 22.4], but also bound the difference between two successive inputs by [0, 0.01], by

```
for (int k=1 ; k<=N ; k++)
  Yk = __BUILTIN_DAED_FGRADIENT(8.42,22.4,0,0.01,8.42,22.4,k,1);
```

Then we get, with the relational domain, much tighter enclosures:

$$
\begin{aligned}
O &= [8.33, 22.5] + [-1.91e^{-4}, 1.91e^{-4}]\varepsilon \\
S &= [-1.18e^{-5}, 0.01] + [-5.34e^{-6}, 5.34e^{-6}]\varepsilon
\end{aligned}
$$

The error is of the same order as previously, but now the bounds for the value of the output are very close to the input bounds, and we get back the information on the variation speed.

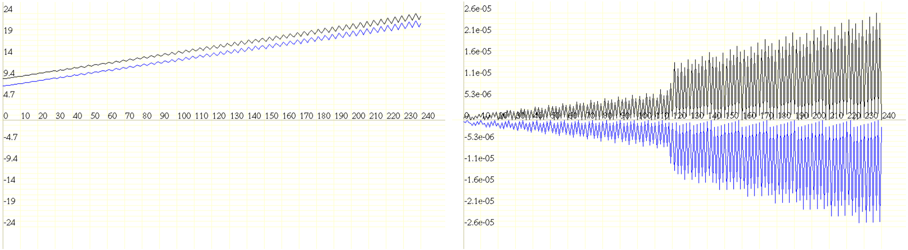
**Complete filter.** We now want to study more closely the behavior of the output. We choose here a plausible scenario for the inputs, that is a sampling of function $i(x)$ defined by

$$
\begin{gathered}
x \leq 0 : i(x) = 1.e^2, \\
x > 0 \ \text{ and } \ i(x) \leq 1.5e^8 : i(x) = i(0) * 2^{50*x/60}.
\end{gathered}
$$

Interval ranges are given for the coefficients of the transformation, and we add a perturbation to the input of the filter thus obtained. We present in Figures 3 and 4, the results got with FLUCTUAT, for the evolution over time of the bounds on the values and errors on the input $Y_k$ of the filter, and of its output. The error on the input is due partly to the logarithm computation, partly to the addition of a perturbation depending on previous inputs. For the time being, we have parameterized the error due to the logarithm computation, which is not yet specified in the IEEE 754 norm.

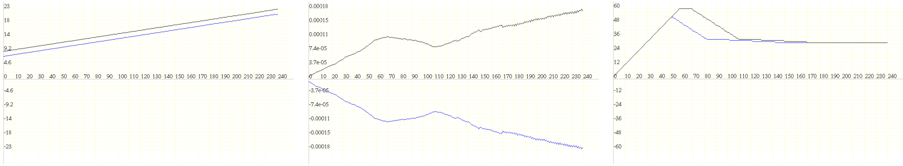
The output is approximately in the same range as the input. We represent in Figure 4 right, the evolution of the depth of the filter: the depth at a given time depends on the values of the parameters and does not have one fixed value. The error on the output is overestimated (no relational analysis for errors here), but we can still observe that the error is not too much amplified. We can also note that its variation is related to the variation of the filter depth.

Finally, we can note that the magnitude of the maximum error on the output, is of the same order as the magnitude on the output obtained with the worst-case scenario. This confirms the relevance of the worst-case analysis.



$Y_k \leq 23.55$ error on $Y_k \leq 2.55e^{-05}$

**Fig. 3.** Evolution of the filter input over time



filter output $\leq 23.02$ error on output $\leq 1.8e^{-04}$ depth of the filter

**Fig. 4.** Evolution of the filter output over time

### 3.3 Linear Filters

Linear filters are a key element of all control systems. We study in this paragraph the outputs $S_i$ of a second order linear filter:

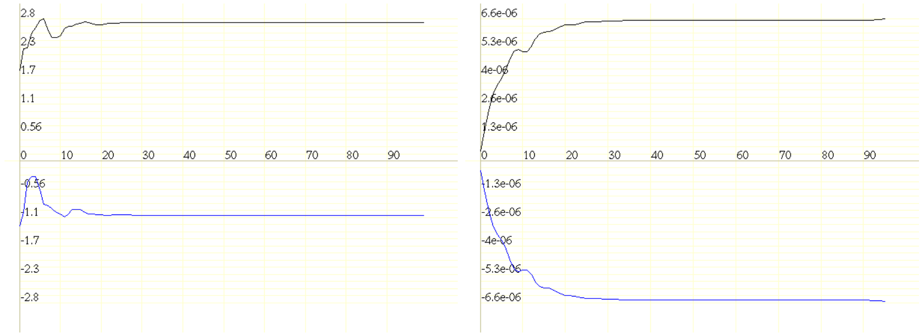
$$S_i = 0.7E_i - 1.3E_{i-1} + 1.1E_{i-2} + 1.4S_{i-1} - 0.7S_{i-2},$$

where $S_0 = S_1 = 0$ and $E_i$ are independent inputs in the range $[0,1]$. These inputs are thus modelled in our relational domain (see section 2.1 or [14]) by

$$\check{E}_i = \hat{E}_i = \frac{1}{2} + \frac{1}{2}\varepsilon_i,$$

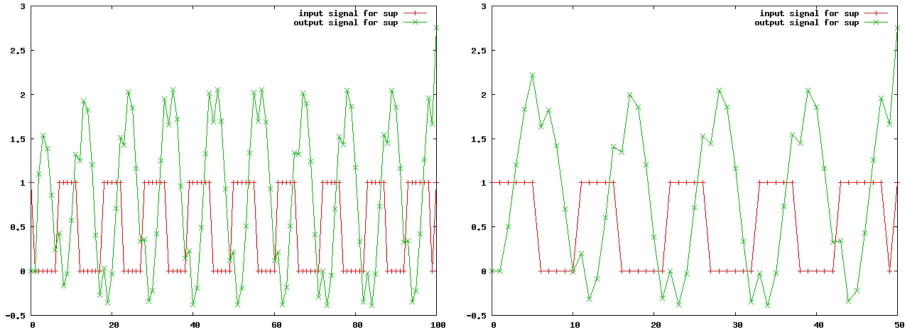
with independent noise symbols $\varepsilon_i \in [-1,1]$. We first consider the output $S_i$ of this filter for a fixed number of unfoldings, e.g. $i = 99$. Fluctuat gives an affine form for the real value of the output $S_i$

$$\hat{S}_{99} = \check{S}_{99} = 0.83 + 7.81e^{-9}\varepsilon_1 - 2.1e^{-8}\varepsilon_2 - 1.58e^{-8}\varepsilon_3 + \ldots - 0.16\varepsilon_{99} + 0.35\varepsilon_{100};$$



Real value of $S_i$ Error on $S_i$

**Fig. 5.** Concretization of $S_i$ for i between 1 and 100



$i = 100$ $i = 50$

**Fig. 6.** Input sequence that maximizes $S_i$, and Output sequence, for $i = 100$ and $i = 50$

whose concretization gives an exact (under the assumption that the coefficients are computed exactly) enclosure $S_{99} \in [-1.0907188500, 2.7573854753]$. The enclosures of the value of successive $S_i$ for $i$ lower or equal to 100 are represented figure 5 (left graph). We also represent on the same figure (right graph) the error on the outputs computed with our new relational domain for the errors, also relying on affine arithmetic.

The coefficients of the affine form allow us to deduce the sequence of inputs $E_i$ leading to the max (or min) of the enclosure of the value of $S_{99}$: take $E_i = 1$ if the corresponding coefficient multiplying $\varepsilon_{i+1}$ is positive, $E_i = -1$ otherwise. For example, the input sequence $E_i$ that maximizes $S_{100}$ is represented figure 6 (left graph), together with the outputs $S_i$, for $i$ between 1 and 100. These sequences, even for such a simple filter, would not be easy to find manually. The input sequence that maximizes $S_{50}$ is represented figure 6 (right graph). This input sequence is fairly different from the first 50 inputs of the input sequence maximizing $S_{100}$: indeed, the successive maximum values of the $S_i$ are not reached for the same sequence of inputs.

Note that the exact enclosure of the filter actually converges towards $S_\infty = [-1.09071884989..., 2.75738551656...]$, and therefore the signal leading to the maximal value of $S_{99}$ is a very good estimate of the signal leading to the maximal value of $S_i$, for any $i \geq 99$.

All this generalizes to linear recursive filters of any order, and can be partly extended to the case of non linear filters (see [15]). The next question to be considered is to find input sequences that maximize the error on the output instead of the value. This is left for future work.

## 4 Conclusion and Future Work

We have shown in this paper how we designed a static analyzer for bounding the imprecision error in numerical programs. This design relied on a careful study of the semantics of the IEEE754 standard, of numerical convergence of the kind of iterative schemes we encountered in control systems, and of specificities of the programming of these systems (SCADE generated code in general). Some real industrial examples were given, which were simple enough to explain in a few pages. The analyzer has already been used (on a low-end PC with 512Mb of memory) for code of the order of 10 thousand lines of C code, and seems to scale up well.

In the future, we plan to invest more on domains dealing precisely with integrators, which make precision analysis hard to carry out, see [12] for example. We also plan to experiment with relational domains for error computations as briefly sketched in [13,16]. We also would like to improve the precision of the least fixed point computation of the abstract equations in our tool, using policy iteration mechanisms, see [6,8]. Last but not least, our abstract domain can be adapted to under-approximations as well, see [15]. A combination of under- and over-approximations will indeed allow us to assess the quality of the results, and give some indications in some cases that the control system under analysis is definitely not implemented in a sufficiently accurate way.

## References

1. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566. Springer, Heidelberg (2002)
2. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A Static Analyzer for Large Safety-Critical Software. In: Proc. PLDI 2003 (2003)
3. Bourdoncle, F.: Abstract Interpretation by dynamic partitioning. Journal of Functional Programming 2(4), 407–435 (1992)
4. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Pottosin, I.V., Bjorner, D., Broy, M. (eds.) FMP&TA 1993. LNCS, vol. 735. Springer, Heidelberg (1993)

5. Comba, J.L.D., Stolfi, J.: Affine arithmetic and its applications to computer graphics. In: SIBGRAPI 1993 (1993)
6. Costan, A., Gaubert, S., Goubault, E., Martel, M., Putot, S.: A policy iteration algorithm for computing fixed points in static analysis of programs. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576. Springer, Heidelberg (2005)
7. Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Symbolic Computation 2(4), 511–547 (1992)
8. Gaubert, S., Goubault, E., Taly, A., Zennou, S.: Static Analysis by Policy Interation on Relational Domains. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421. Springer, Heidelberg (2007)
9. Goubault, E., Guilbaud, D., Pacalet, A., Starynkévitch, B., Védrine, F.: A Simple Abstract Interpreter for Threat Detection and Test Case Generation. In: WAPATV 2001 (2001)
10. Goubault, E.: Static analyses of the precision of floating-point operations. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126. Springer, Heidelberg (2001)
11. Goubault, E., Martel, M., Putot, S.: Asserting the precision of floating-point computations: a simple abstract interpreter. In: Le Métayer, D. (ed.) ESOP 2002 and ETAPS 2002. LNCS, vol. 2305. Springer, Heidelberg (2002)
12. Goubault, E., Martel, M., Putot, S.: Some future challenges in the validation of control systems. In: Proceedings of ERTS 2006 (2006)
13. Goubault, E., Putot, S.: Weakly Relational Domains for Floating-Point Computation Analysis. In: NSAD 2005 (2005)
14. Goubault, E., Putot, S.: Static Analysis of Numerical Algorithms. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134. Springer, Heidelberg (2006)
15. Goubault, E., Putot, S.: Under-approximations of computations in real numbers based on generalized affine arithmetic. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634. Springer, Heidelberg (2007)
16. Goubault, E., Putot, S.: Automatic analysis of imprecision errors in software (2007), http://www.di.ens.fr/~goubault/papers/abstract.pdf
17. Jones, N.D., Muchnick, S.S.: A flexible approach to interprocedural flow analysis and programs with recursive data structures. In: POPL 1982 (1982)
18. Landi, W., Ryder, B.: A safe approximate algorithm for inter-procedural pointer aliasing. In: Proceedings of PLDI. ACM, New York (1992)
19. Martel, M.: Propagation of roundoff errors in finite precision computations: a semantics approach. In: Le Métayer, D. (ed.) ESOP 2002 and ETAPS 2002. LNCS, vol. 2305. Springer, Heidelberg (2002)
20. Miné, A.: Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986. Springer, Heidelberg (2004)
21. Putot, S., Goubault, E., Martel, M.: Static Analysis-Based Validation of Floating-Point Computations. In: Alt, R., Frommer, A., Kearfott, R.B., Luther, W. (eds.) Numerical Software with Result Verification. LNCS, vol. 2991. Springer, Heidelberg (2004)
22. Stolfi, J., de Figueiredo, L.H.: An introduction to affine arithmetic. TEMA Tend. Mat. Apl. Comput. 4(3), 297–312 (2003)
23. Goubault, E., Putot, S.: Fluctuat user manual (2007) (available upon request)
24. Grammatech Inc. CodeSonar, overview, http://www.grammatech.com/products/codesonar/overview.html
25. PolySpace Technologies. PolySpace for hand-written code, http://www.polyspace.fr/products.htm
26. LIP6. The CADNA Library, http://www-anp.lip6.fr/cadna/Accueil.php